Viceroy: A Scalable and Dynamic Emulation of the Butterfly

Dahlia Malkhi[*]
School of Computer Science
and Engineering
The Hebrew University
Jerusalem 91904, ISRAEL
dalia@cs.huji.ac.il

Moni Naor[†]
Dept. of Computer Science
and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, ISRAEL
naor@wisdom.weizmann.ac.il

David Ratajczak
University of California
Berkeley, CA 94720, USA
dratajcz@cs.berkeley.edu

ABSTRACT

We propose a family of constant-degree routing networks of logarithmic diameter, with the additional property that the addition or removal of a node to the network requires no global coordination, only a constant number of linkage changes in expectation, and a logarithmic number with high probability. Our randomized construction improves upon existing solutions, such as balanced search trees, by ensuring that the congestion of the network is always within a logarithmic factor of the optimum with high probability. Our construction derives from recent advances in the study of peer-to-peer lookup networks, where rapid changes require efficient and distributed maintenance, and where the lookup efficiency is impacted both by the lengths of paths to requested data and the presence or elimination of bottlenecks in the network.

1. INTRODUCTION

The Viceroy network construction tackles a fundamental problem of practical distributed systems: the discovery and location of data and resources in a dynamic network. It functions as a distributed hash table (DHT), managing the distribution of data among a changing set of servers, and allowing clients to contact any participating server in the network to find any stored resource by name. It is primarily intended for environments in which the scale and dynamism are so great as to require performance metrics outside the scope of classical routing network research.

The motivation for our research stems from the recent emergence of peer-to-peer applications on the Internet, where it has become apparent that in order to share resources and access services over large, dynamic networks, users must first have the means to locate them in an efficient manner. At an abstract level, these applications all implement or employ a distributed hash table that maps names to values and that functions as a supporting lookup service. The Domain Name System (DNS) is a known example of such a lookup service, but one that is static and that suffers from congestion problems at the root of the DNS tree. In contrast, we aim to build a completely distributed and scalable lookup service suitable for deployment in peer-to-peer networks, where the set of participating servers is particularly dynamic and no central control or information is easily maintained.

There are two main challenges that large-scale DHTs must overcome: the first is distributing data in such a way that it remains nearly balanced across the set of active servers, and such that only small changes are necessary when servers join and leave. The second is to maintain a network of connection information between servers so that a lookup for data can be “routed” between servers toward its intended target, and so that servers may join and leave without requiring hash information to be propagated through the entire network. The Viceroy algorithm manages the first problem by employing consistent hashing [5] in a manner nearly identical to Chord [20]. However, Viceroy improves upon existing solutions to the second problem by maintaining a connection graph that is a constant degree, logarithmic diameter approximation to a butterfly network.[1]

Objectives: We are primarily interested in Viceroy’s behavior in networks of extreme scale, dynamism, and traffic. It must be able to cope with large volumes of client lookups, thus we wish to ensure that there are no bottlenecks through which lookups stagger. The joining or leaving of a single server should not impair the availability of the service, so we want such changes to have only a small and mostly local impact on the network. We capture these design goals with the following performance metrics:

Congestion: No server should be a bottleneck on the performance of the service. The load incurred by lookups routing through the system should be evenly distributed

[*] Supported in part by an Israeli Ministry of Science grant #1230-3-01

[†] Supported in part by an IST Grant RAND-APX.

PODC 2002 Monterey, California, USA

[1] We confine ourselves to this second aspect of Viceroy, and we refer the reader to the above citations for a more thorough analysis of consistent hashing, its many desirable qualities, and the subtleties involved with a distributed implementation.

among participating lookup servers. We define this formally in Section 6.1.

Cost of join/leave: The service should accommodate changes easily. In particular, when servers join or leave, only a small number of servers should change their state.

Lookup path length: The forwarding path of a lookup should involve as few machines as possible. We aim to minimize the maximum path length in the network. In later sections, this will be referred to as the *dilation* of the network.

Clearly there are tradeoffs associated with some of these objectives. Creating a network with greater connectivity may reduce lookup path length, but it will also increase the number of linkage updates during joins and leaves; on opposite extremes would be a simple ring, and a complete graph. Similarly, there are some schemes that optimize all criteria but one. A balanced binary tree[2], for example, has desirable locality properties and logarithmic path length, but has abysmal congestion at the root.

The Case for constant degrees: In developing Viceroy, our primary motivation was to achieve an algorithm with constant linkage cost, logarithmic path length, and with the best achievable congestion given those constraints. We emphasize constant linkage cost for several reasons. First, we believe that the practical cost of updating links far exceeds normal lookup costs because it involves coordination between multiple machines; each change could require new session information to propagate between machines, or could even require multiple machines to acquire locks in order to rigidly maintain the consistency of the network. Second, reducing the number of edges in the graph reduces the ambient traffic associated with pings and control information, and maintaining a constant degree network assuages concerns about the cost of too many open connections at servers. Third, there are cases when it is desirable for a server to notify its outgoing and incoming connections that it is leaving, and thus the degree of the network directly relates to the load incurred by joins and leaves.[3] While it is a justifiable concern that low degree networks are unsuitable for failure-prone environments, we feel that issues of fault tolerance should be handled separately from routing design choices using clustering techniques. Section 7.2 describes one such scheme and presents further justification for making this design choice.[4]

The framework: In order to distribute key-value pairs across a set of participating servers, we treat both the servers and the keys as points in a specially chosen metric where two keys that are *close* under the metric reside on servers that are also *close* under the metric. A particular key-value pair resides on the server that is *closest* to the key. The choice of the next hop in a route is calculated based on a comparison between the point of the target key and the current server's point. This framework captures several recent proposals, including Consistent Hashing [5] and Chord [20] (rings), CAN [18] (d-dimensional tori), Plaxton et al. work [16] and OceanStore [7] (hypercubes). (See more in the Section 1.1.) The mapping we use is identical to Chord as we map keys and servers to the unit ring [0..1). For a given set of active servers, a server manages all key-value pairs that have names between its counter-clockwise neighbor's name and its own name. We therefore say that a key is mapped to its *successor* on the ring.

Routing on Viceroy networks uses links between *successors* and *predecessors* on the ring for short distances. Borrowing from the ideas of Kleinberg [6] and Barriere et al. [2], we augment the ring construction with a constant number of *long range contacts* chosen appropriately so that a localized routing strategy produces short paths. The inspiration from [6] is that these few long range contacts should be chosen randomly with a particular bias toward closer points.[5] In attempting to improve the worst-case performance of such a scheme, we arrived at an approximation to the classical butterfly network, as described below. Unlike Kleinberg's work, the resulting topology has only logarithmic dilation (instead of polylogarithmic).

The Viceroy topology: Viceroy networks are a composition of an approximate butterfly network[6] and the connected ring of predecessor and successor links mentioned previously. In addition to predecessor and successor links, each server includes five outgoing links to chosen long range contacts. First, each node selects a *level* at random in such a way that when n servers are operational, one of $\log n$ levels is selected with nearly equal probability. For a level ℓ node, two edges are added connecting it to nodes at level $\ell + 1$: A "down-right edge is added to a long-range contact at level $\ell + 1$ at a distance roughly $1/2^\ell$ away, and a "down-left edge at a close distance on the ring to level $\ell + 1$. In addition, an "up" edge to a close-by node at level $\ell - 1$ is included if $\ell > 1$. Finally, "level-ring" links are added to the next and previous nodes of the same level ℓ. Figure 1 depicts an ideal Viceroy network. For simplicity, the up link, as well as ring and level-ring links are not depicted.

Routing proceeds in three stages: the first one consists of a climb using up connections to a level-1 node. In the second stage, routing proceeds down the levels of the tree using the down links; moving from level ℓ to level $\ell + 1$, one follows either the edge to the close-by down link or the far-away down link, depending on whether x is at distance greater than $1/2^\ell$ or not. This continues until a node is reached with no down links, which presumably is in the vicinity of the target. Finally, a "vicinity" search is performed using the ring and level-ring links until the target is reached (which may not be at a leaf in the tree). We will show that this process requires only $O(\log n)$ steps with high probability in a random network construction[7]. Formalizing and proving this result, as well as measuring the other performance metrics, will be the focus of the remainder of this paper.

Contributions of this work: We present a simple and ef-

[2] specifically, those variants with constant linkage cost [9, 14]

[3] In [8], it is shown that such communication is necessary to construct a DHT with atomic data semantics.

[4] See [8] for a more thorough examination of this approach.

[5] While Kleinberg's paper is descriptive, explaining how a social network allowing efficient routing may develop, a construction readily follows from his work.

[6] A butterfly network is described, e.g., in [19].

[7] Throughout this work we use the term *with high probability* (w.h.p.) to mean with probability at least $1 - c/n$ for some $c > 0$.

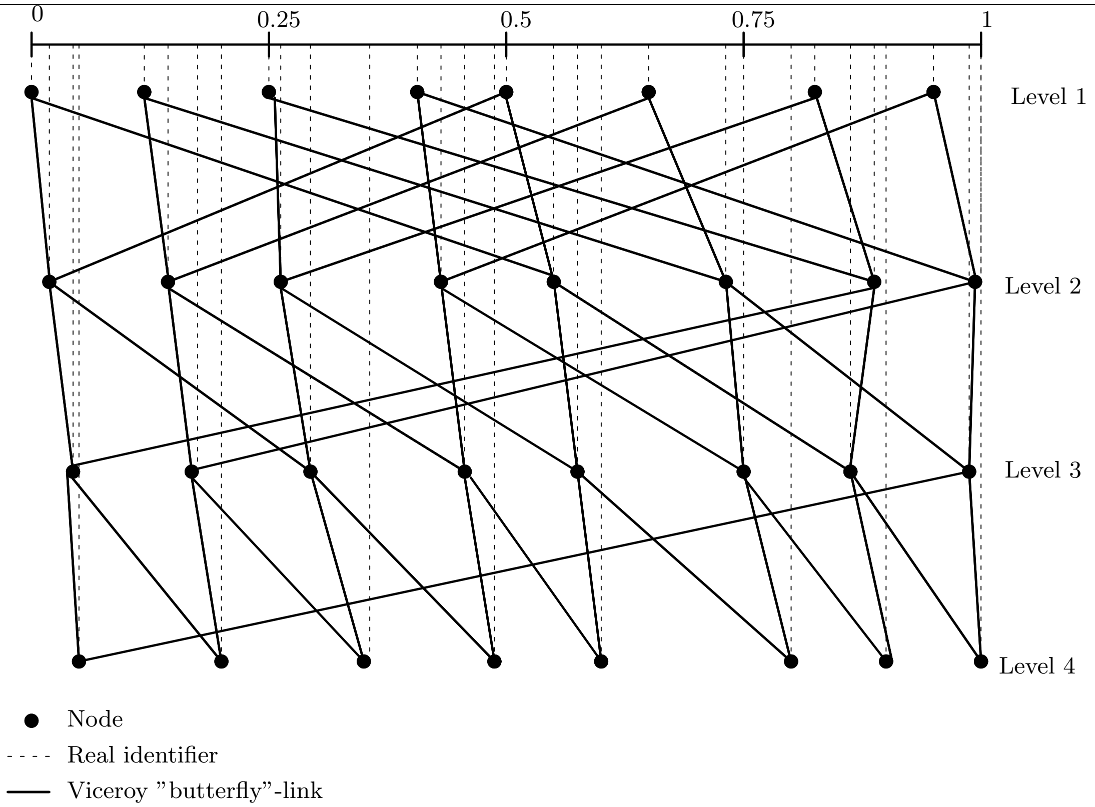

Figure 1: An ideal Viceroy network. Up and ring links are omitted for simplicity.

ficient network construction that maintains constant degree networks in a dynamic environment. Routing is achievable on these networks in $O(\log n)$ hops and with nearly optimal congestion. This is the first proposal of a decentralized and dynamic network where all of these properties are achieved simultaneously. The events of a server joining and leaving the system induce $O(\log n)$ hops and require only $O(1)$ servers to change their states.

1.1 Related work

The proliferation of Internet-scale services and the advent of peer-to-peer applications has focused considerable attention on the resource lookup problem presented here. Many popular services, such as DNS, LDAP, and file-sharing services (such as Napster and Gnutella), rely on some partitioning of data across a dynamic network. Recently, several schemes have been proposed that address the scalability and dynamic requirements of a world-wide lookup service.

The Chord lookup service [20] presents a solution that has greatly influenced our approach. Specifically, we borrow from it the design that maps name servers and resources onto the same domain, and the use of a doubly-connected ring as the underlying structure. However, our solution differs from it in several important ways. First, in Chord, each node maintains a logarithmic number of long-range links, yielding a logarithmic number of join/leave linkage updates in expectation and with high probability. In contrast, Viceroy has constant out-degree, yields a constant expected linkage cost, and with high probability, a random sequence of $O(\log n)$ updates has a total $O(\log n)$ linkage cost. Second, in Chord, the way in which the network is maintained appropriately over the lifetime of the system is not fully described, and is deferred to a background maintenance process. In contrast, we have placed an emphasis on fully describing the way in which the network evolves during server joins and leaves, and on analyzing its worst case behavior.

The algorithm of Plaxton et al. [16] was originally devised to route web queries to near-by caches, and has been employed in the Tapestry naming service [21]. Similar to Chord, its basic scheme is a distributed hash table with efficient request routing between lookup servers using an efficient, dynamic routing structure. The routing algorithm is a randomized approximation of a hypercube. Compared with our scheme, the method of [16] has logarithmic expected join/leave complexity.

The recent work by Ratnasamy et al. [18] places a similar stress to ours on a constant degree network for routing lookup requests. Their *Content Addressable Network* (CAN) dynamically maintains an approximation to a d-dimensional torus, for a chosen constant d. Their expected routing complexity is $O(dn^{(1/d)})$, compared with logarithmic in our construction.

Table 1 provides a snapshot of the performance of each of these methods, as well as the Viceroy scheme, according to our chosen performance metrics.

Work on data structures has produced some related results. SkipLists [17], for example, lend themselves to distributed construction and similarly employ randomization for efficient routing. Compared with our method, some of the nodes in a SkipList suffer a load which is linear in the

Lookup scheme	dilation	congestion	linkage
Chord[20]	$\log n$	$(\log n)/n$	$\log n$
Tapestry[21]	$\log n$	$(\log n)/n$	$\log n$
CAN[18]	$dn^{(1/d)}$	$dn^{(1/d-1)}$	d
Small Worlds [6]	$\log^2 n$	$(\log^2 n)/n$	$O(1)$
Viceroy (ours)	$\log n$	$(\log n)/n$	7

Table 1: Comparison of expected performance measures of lookup schemes.

total traffic, where the load in our construction is balanced. Certain types of constant linkage-cost data structures are competitive with our scheme in every way except for congestion [9, 14]. Proposals for distributed data structures typically address the problem of join and leave complexity and state partitioning, but either require centralized control, or do not address congestion [4, 12].

There are other works that have looked at the problem of dynamic network construction with different emphases than ours. Pandurangan et al. address in [15] the problem of dynamically constructing a low-degree, logarithmic diameter network under a probabilistic model of arrival and departure; however their construction does not provide a routing scheme and their intended application is to disseminate queries to every server rather than to route to a particular server. Their method is also not fully distributed and employs a central server for newly connected nodes. One can view Viceroy as an improvement over their scheme in that logarithmic diameter is achieved in a distributed fashion, and its desirable congestion properties means that Viceroy can also be used for dissemination. Lin et al. are concerned in [10] with building graphs that remain connected in the face of many failure scenarios. Fiat and Saia in [3] are concerned with constructing a logarithmic diameter routing network that remains connected despite attacks on a large fraction of its nodes. Both of these constructions are static, and have logarithmic degree at each node.

2. SYSTEM MODEL AND PRELIMINARIES

2.1 Terminology and notation

The system consists of a varying set of participating servers (or simply, servers). We sometimes refer to an active set of servers as a *configuration*. Servers have real identifiers chosen uniformly at random from the range [0..1) before they join. By slight abuse of notation, we will refer to server identifiers simply as the servers themselves. All arithmetic in this range will be done modulo 1, and we refer to it as the *ring*. For any three real values $x, y, z \in [0..1)$, $stretch(x, y)$ is the clockwise region between x and y non-inclusive, and z is *between* x and y if $z \in stretch(x, y)$. There are two measures of interest related to stretches: their length and the *number* of servers in them: The distance $d(x, y)$ is the clockwise distance from x to y. The density $q(x, y)$ is the number of servers present in $stretch(x, y)$.

We denote by $\text{SUCC}(x)$, the clockwise neighbor of x on the ring in the current configuration, and $\text{PRED}(x)$, its counter-clockwise neighbor. Also, as a matter of convenience, we will use the term clockwise-closest as a shorthand for “closest in the clockwise direction”

Each server s has an additional positive integer identifier called its *level*, denoted $s.level$. The meaning of levels will become apparent when we discuss our network construction, but it intuitively indicates a server's placement in the butterfly network we construct. We denote by $\text{NLEVEL}_i(x)$, the clockwise-closest level-i server to x, i.e., a server of level i that is closest in the clockwise direction to x. Similarly, $\text{PLEVEL}_i(x)$ denotes the server of level i that is closest in the counter-clockwise direction to x. We define the shorthand $\text{NEXTONLEVEL}(x) = \text{NLEVEL}_{x.level}(x)$, i.e., the closest server of the same level as x in the clockwise direction from x (likewise, $\text{PREVONLEVEL}(x) = \text{PLEVEL}_{x.level}(x)$). Finally, the level-density $q_i(x, y)$ denotes the number of level-i servers present in $stretch(x, y)$.

2.2 System model and assumptions

Each server maintains certain information about other servers to facilitate routing and the distribution of data in the network. When we say that a server s has a link to s', this means that s has established a connection to s' and may forward requests to it. In practice this means that physical address information must be known to each of them, and internal state is kept about incoming and outgoing connections. For our purposes we will assume that any server can establish a connection to any other if it knows that server's identifier, and merely say that a “connection is made” rather than elaborate on how this is done at the protocol level.

Any server may voluntarily leave the system and new servers may spontaneously join it. However, we assume that joins and leaves are *not* correlated with server identifiers, and thus the server identifiers of an active set of servers always appear randomly distributed about the ring. In this paper we assume that multiple join and leave operations do not overlap, and servers never fail; we refer the reader to [8] for techniques to achieve concurrency and to handle failures in the system. In principle the system is capable of handling many parallel joins and leaves since with high probability the servers that have to change their state do not overlap; however to avoid dealing with issues such as locking we made the above assumption.

Lastly, it is an important feature that a client may invoke a lookup operation at any active server. However, the mechanism by which a client or joining server is initially advised of an active server is not discussed here.

3. THE VICEROY NETWORK

Every participating server has two associated values that determine its connectivity to the rest of the system: its *identity* $s.id \in [0, 1)$ and its *level* ℓ which is a positive integer. The identity of a server is fixed throughout its participation in the system, but the level will sometimes be changed in the course of the network's evolution. The link and routing structure of a Viceroy network is determined entirely by the identity and level information of the currently active servers, and is not affected by historical configurations or other sources of randomness.

Remark: given that the identity is fixed, we will simplify our notation and refer to both the server and the identity as s (and drop the $s.id$).

The network consists of three sets of links: a *general ring*, where each node is connected to its successor and predecessor, *level rings*, where nodes of the same level are connected to each other in a ring, and the *butterfly* where each non-leaf node at level ℓ points to two "down" nodes of level $\ell+1$ and each node at level $\ell > 1$ points to an "up" level $\ell - 1$ node. The left down link is chosen to be the first node of level $\ell+1$ clockwise-closest to s (i.e. $\text{NLEVEL}_{\ell+1}(s)$), and the right down link is chosen to be the first node of level $\ell+1$ clockwise-closest to $s + 1/2^{\ell}$ (i.e. $\text{NLEVEL}_{(\ell+1)}(s + 1/2^{\ell})$). An up link is chosen to be the level $\ell - 1$ node clockwise-closest to s, i.e. $\text{NLEVEL}_{\ell-1}(s)$. Figure 1 depicts the relationship between the ring id's and the butterfly links.

It remains to discuss how levels are chosen, how joins and leaves affect this construction, and how routing is performed along the established connections.

4. IDENTITY AND LEVEL SELECTION

The ideal conditions for our algorithm occur when an equal fraction of n servers select each level between 1 and $\log n$, and the server identifiers at any particular level are evenly distributed about the ring. Achieving these conditions exactly, however, would require a knowledge of $\log n$ and might require all server identifiers and possibly all levels to change whenever a server joins or leaves. Without precise knowledge of n, and without altering server identifiers, our goal is to select levels that approximate the ideal conditions in a way that requires as few level changes as possible when a join or leave occurs. In this section we examine the properties of the random identity selection, and describe a way of selecting levels that achieves the above goal.

In our model, each server picks its identifier independently and uniformly from $[0, 1)$ and this id does not change while it is active in the system. In practice, we are not concerned with choosing an id with infinite precision, and it suffices to generate a fixed number of random bits so as to plausibly avoid collisions.[8]

We now describe a distributed level selection algorithm and show that it provides a good dispersal of levels among servers. The basic idea is as follows: a server s first estimates n, then based on this estimate, n_0, it chooses a level between $1 \dots \lfloor \log n_0 \rfloor$ uniformly at random. Of course, the best way to estimate n is to simply count the servers that exist at the time of estimation. However, since precisely accounting the size of the network would require information to be propagated to every server each time the size changes, this would be too costly. Our goal is then to provide a good estimate of n while keeping the estimation procedure reasonably localized. To this end, our estimation procedure only utilizes the identifier information of the servers in the vicinity of the estimating server, s.

```
SELECT-LEVEL($s$):

  1. Let $n_0 = 1/d(s, \text{SUCC}(s))$.

  2. Select a level $\ell$ among $[1 \dots \lfloor \log n_0 \rfloor]$ uniformly at random and return $\ell$.
```

Figure 2: Level selection algorithm

Our simple algorithm is presented in Figure 2. By noticing that the expected distance between the estimating server and its successor is roughly $1/n$, we can get a "ballpark" estimate for n. The elegance of this scheme is that a server must only reselect a level if its successor changes.[9].

Good and sane networks: Our identity and level selection is designed so as to achieve a good dispersal of servers and level within the ring. The precise properties we need, sanity and goodness, are as follows (Recall that for servers s, t, $q(s,t)$ denotes the number of servers in $\mathit{stretch}(s,t)$ and $q_i(s,t)$ denotes the number of servers of level i in $\mathit{stretch}(s,t)$):

Sanity: When n servers are present, then w.h.p. every server running the SELECT-LEVEL algorithm estimated $\log(n/(2\log n)) \leq \lfloor \log n_0 \rfloor \leq 3 \log n$. We say that any level $\ell < \log(n/(2\log n))$ is "sane."

Goodness: When n servers are present, the following is satisfied:

(1) A logarithmic-length stretch contains a logarithmic number of servers: For all servers s, w.h.p. $q(s, s + (\log n)/n) = O(\log n)$. **(2)** Finding a server on a particular level takes a logarithmic number of ring steps in expectation, and square-log at worst: For all servers s and for $i \leq \log(n/(2\log n))$, in expectation $q(s, \text{NLEVEL}_i(s)) = O(\log n)$ and w.h.p. $q(s, \text{NLEVEL}_i(s)) = O(\log^2 n)$.

(3) Finding a server on a 'sane' level $j \leq \log(n/(2\log n))$ takes a logarithmic number of ring steps: For all servers s, w.h.p. $\min_{j \leq \log(n/(2\log n))}\{q(s, \text{NLEVEL}_j(s))\} = O(\log n)$.

(4) A stretch of logarithmic length contains one server in expectation, and at least one with non-negligible probability: For all servers s and for $i \leq \log(n/(2\log n))$, in expectation $E[q_i(s, s + (3\log n)/n)] = 1$ and with probability at least $1/2$ it holds that $q_i(s, s + (3\log n)/n) \geq 1$.

(5) Level selections are evenly interleaved to within a logarithmic factor: For all servers s and for $i, j \leq \log(n/(2\log n))$, in expectation $E[q_i(s, \text{NLEVEL}_j(s))] = 1$ and w.h.p. $q_i(s, \text{NLEVEL}_j(s)) = O(\log n)$.

In the following claim, we state the sanity and goodness of our construction. The proof technique is rather standard, and is omitted due to space constraints.

CLAIM 4.1. *The network identity and level selection scheme above is sane and good.*

5. VICEROY CONSTRUCTION AND ROUTING

In this section, we describe how a Viceroy lookup network is constructed and maintained dynamically in a completely decentralized manner. We describe the operations: JOIN and LEAVE, which both call LOOKUP as a subroutine. We will discuss two variants of the LOOKUP operation in the next section.

A server s maintains the following variables, initially undefined:

$s, s.\mathit{level}$**:** the identity and the level chosen by s.

$s.\mathit{predecessor}, s.\mathit{successor}$**:** ring pointers.

[8] In any case, collisions can be detected by merely performing a lookup on the chosen identity.

[9] We can improve this further by saying that a new level must be chosen only when $\lfloor \log n_0 \rfloor$ changes (otherwise the current random selection is good enough). Also, even if $\lfloor \log n_0 \rfloor$ changes, the new level chosen must only be put in place if it was a level that didn't exist before (if n_0 increases), or if the current level no longer exists (if n_0 decreases)

s.nextonlevel, *s.prevonlevel*: level ring pointers
s.left, *s.right*, *s.up*: butterfly pointers.

JOIN **Operation:** The main operation we describe is how a server joins the network. For a server s to join the network, it performs the steps in Figure 3. If a server's level changes, then it will have to repeat steps 4 and 5.

1. Select an identity s according to the identity selection mechanism described in Section 4.
2. Find using the LOOKUP subroutine SUCC(s). Update *s.predecessor*, *s.successor*, PRED(s)*.successor* and SUCC(s)*.predecessor* so as to insert s in its place on the ring.
3. Transfer from the successor all key-value pairs with key between *s.predecessor* and s.
4. Select a level *s.level* according to the level selection mechanism of Section 4. Find (by single stepping on the ring) $s' = \mathrm{NEXTONLEVEL}(s)$ and $s'' = \mathrm{PREVONLEVEL}(s)$. Update *s.nextonlevel*, *s.prevonlevel*, s''*.nextonlevel*, and s'*.prevonlevel* to insert s into its place on the level ring.
5. Find $\mathrm{NLEVEL}_{(s.level+1)}(s)$ (by single-stepping in the clockwise direction) and assign it to $s.left$. Find (using LOOKUP) clockwise-closest$(s + 1/2^i)$, and then find (by single stepping in the clockwise direction) $\mathrm{NLEVEL}_{(s.level+1)}$(clockwise-closest$(s + 1/2^i)$). Assign it to *s.right*. Find $\mathrm{NLEVEL}_{(s.level-1)}(s)$ and assign it to *s.up*.

Figure 3: JOIN **algorithm.**

LEAVE **Operation:** When a server leaves, it will have to remove all of its outbound connections, and notify all of the servers with inbound connections that they must find a replacement (which they will find using the LOOKUP subroutine)[10]. It must also transfer its resources to its successor.

Because of our level selection algorithm, it is likely that at levels beyond $(\log n)/2$ there may be a sufficient gap between servers at the same level that single-stepping between them would take an unreasonable amount of time. In order to account for this, as well as to account for the possibility of certain levels not existing, when choosing up and down links as well as level ring links, we leave them unspecified if it requires more than $O(\log^2 n)$ steps after the initial LOOKUP to find them. Note that by the goodness properties of level selection, none of the levels up to $(\log n)/2$ are affected by this detail.

6. SIMPLE LOOKUP

In this section we introduce a slightly simplified routing protocol that performs competitively in the average case and illustrates the basic principles that we will refine in Section 7 to yield stronger high probability bounds. The simplified routing does not use the level ring links, only the successor, predecessor, and butterfly links.

Generally, the lookup subroutine is used both for maintaining the structure of the network (updates during join and leave) and for looking up keys. The purpose of subroutine LOOKUP(x, y) is to find, starting at server y, the server clockwise-closest to the value x.

Routing occurs in three distinct phases. In the first phase, a level-1 ("root") server is found by following up links. In the second phase, the lookup is routed down from the root along down connections. In this phase, when we are at a node on level i we are at a distance at most $1/2^{i-1}$ from the target. Then if the target is at distance $1/2^i$ or greater, we traverse down through the right link, and otherwise, down the left link. This phase terminates when it reaches a node with no down links or a node that overshoots the target. In the third phase, the ring is traversed either in the clockwise direction or in the counter-clockwise direction to the desired target. Figure 4 describes this procedure.

A couple of things are worth pointing out: The first and second stages will also be used in our more sophisticated lookup scheme, though the third phase will be different. A distributed version of this lookup algorithm would require some state denoting the current phase to follow a lookup request. It would also require a choice on how the return value is returned to the originator of the lookup (either directly, or by backward-chaining along the lookup path).

LOOKUP(x, y):

Initialization: Set *cur* to y.

Proceed to root: If $cur.level = 1$ goto traverse-tree phase.
Else set $cur = cur.up$ if it exists or else, $cur = cur.successor$ and repeat phase.

Traverse tree: If $d(cur, x) < 1/2^{cur.level}$ then $cur = cur.left$ (go down left) if it exists.
If $d(cur, x) \geq 1/2^{cur.level}$ then $cur = cur.right$ (go down right) if it exists.
If the required down link does not exist or if it exists and it overshoots the target, then goto traverse-ring phase. Else repeat this phase.

Traverse ring: If *cur* is the clockwise-closest server to x then compute a response to the lookup. Else, set $cur = cur.successor$ or $cur = cur.predecessor$, whichever is closer to x and repeat phase.

Figure 4: Simple LOOKUP **algorithm.**

6.1 Simple Viceroy analysis

In this section we make a preliminary analysis of the Viceroy lookup network according to the performance metrics introduced in Section 1. Recall that the definition of sanity (Claim 4.1) guarantees that w.h.p. all levels are under $3 \log n$. This immediately implies that traversing up and down the tree takes $O(\log n)$ steps:

LEMMA 6.1. *If n servers are present, then the first two phases take $O(\log n)$ steps w.h.p. (over the identifier and level choice).*

[10] This can be performed more efficiently by noticing that the replacements will point to the successor on the general or level rings, unless a level changes.

Proof. The main point to notice here is that starting with a node on level higher than $\log(n/(2\log n))$, there may not exist up links to follow. However, by the goodness property (3) (Claim 4.1), traversing successors in the 1st phase finds a node with level $\leq \log(n/(2\log n))$ in $O(\log n)$ steps. From there, each step in the first phase decreases the level by one, so the first phase takes $O(\log n)$. By sanity, there are at most $3\log n$ network levels, and since each step in the second phase increases the level by one, it also takes $O(\log n)$. □

The third phase involves single stepping along ring links. In this simple version, it takes $O(\log n)$ steps in expectation, and at worst $O(\log^2 n)$ steps. Below, we optimize this step to use level ring links, and bring the worst case cost down to $O(\log n)$.

Lemma 6.2. *If n servers are present, then the last phase takes an expected $O(\log n)$ steps and $O(\log^2 n)$ steps w.h.p. (over the identifier and level choice.)*

Proof. In the "Traverse-tree" phase, each time a down link is followed, the maximum possible distance to the target is halved. Therefore, at level k the greatest possible distance to the target is 2^{1-k}. By goodness property (2) (Claim 4.1), down links exist for all nodes at level $k \leq \log(n/(2\log n))$. So when a node s at level $\log(n/(2\log n))$ is ever reached in this phase, the maximum distance from to the target is $(12\log n)/n$, which by goodness property (1) (Claim 4.1) is $O(\log n)$ steps away from the target; or it might overshoot the target by an expected $O(\log n)$ servers, and w.h.p. at most $O(\log^2 n)$ servers (again by goodness property (2) Claim 4.1). Therefore, we have that the last phase takes an expected $O(\log n)$ and at worst $O(\log^2 n)$ total single steps. □

Since the above lemmata hold regardless of the originating server or the target server, we get the following as a result:

Corollary 6.3. *If n servers are present in a Viceroy network, then the expected (over source and destination) dilation of the* simplified *lookup scheme is $O(\log n)$ and it is $O(\log^2 n)$ w.h.p. (over the identifier and level choice.)*

We define the *load* of a lookup on a server to be the probability that it is involved in a search for a random value from a random starting point; the congestion of a network is then the maximum load on any server. This measure gives us an indication of whether there is a substantial bias in the number of routes through particular nodes during normal random traffic (where the assignment of keys is considered sufficiently random and there are no "hotspots")[11] and during periods of heavy traffic when nearly every route is being traversed. Given that we have a bounded degree network, $\Omega(\log n)$ servers take part in a lookup for a random value and thus the best load we can expect would be $\Omega(\log n/n)$. On the other hand we have:

Theorem 6.4. *If n servers are present in a Viceroy network, then for any server the expected load is $O((\log n)/n)$ and w.h.p. the maximum load on all servers is $O((\log^2 n)/n)$.*

Proof. To evaluate the load we will consider three possibilities for a server s to participate in a lookup from server y to x, corresponding to the three phases of routing. It is straight forward to see that each phase induces an expected $O(\log n/n)$ load on s. We move to bound the worst case load.

Our proof strategy is to concentrate on a single type of walk leading to s, whereby type of walk we mean any particular, either upward or downward, sequence of link selections (e.g., left-left-left-right-left). In order to bound the worst case, we will prove below, in Lemma 6.5, that all nodes leading to s using a particular type of walk are within distance $O(\log^2 n)$ of one another, w.h.p. We will make use of this property to bound the three phases separately as follows:

Node s is on the path to root (phase 1) Looking at the first phase, for a node s to be on the path to root from y, it must be on an upward path (e.g., up-up, or up-up-up). Since the strict distance of all upward links is zero, all sources y leading to s on upward paths are within distance $O(\log^2 n)$ from one another w.h.p. Therefore, out of all paths, w.h.p. at most $O((\log^2 n)/n)$ go through s in phase 1.

Node s is on route from level 1 to the lowest level The proof here is divided into two parts. First, we "count" the number of root nodes that have a path that goes through s. Then we calculate the probability that each such path is used.

By Lemma 6.5, for each particular type of walk from level 1 to s, there are source roots within at most $O(\log^2 n/n)$ distance from one another w.h.p. We now turn to the probability that a particular walk from any root is chosen. By assumption, targets are distributed uniformly on the ring. Therefore, a server s of level i receives at most $1/2^{i-1}$ of the walks that go through a particular root. In fact, the load may be somewhat lower due to overshooting, at which point the butterfly path is abandoned and the ring is traversed.

Putting all of the above together, we have that a node s of level i is on 2^i types of walks from roots. Each has $O((\log^2 n)/n)$ load w.h.p., and is selected with probability $1/2^{i-1}$. This induces a load of $O((\log^2 n)/n)$ on s w.h.p.

The final routing stage This stage involves single stepping on the ring for an expected $O(\log n)$ steps and at worst $O((\log^2 n))$ w.h.p. by Lemma 6.2. Hence, there are an expected $O(\log n)$ source nodes and $O(\log^2 n)$ nodes w.h.p. on such paths.

In total, among all randomly selected sources and targets, there are expected $O(\log n/n)$ routes that go through s, and $O(\log^2 n/n)$ routes w.h.p., and this is the load inflicted on it.

□

Lemma 6.5. *Let s be a node, and let two nodes r_1 and r_2 be such that a particular type of upward or downward "walk" (e.g., left-left-right) leads from both r_1 and r_2 to s. Then r_1 and r_2 are within distance $O(\log^2 n/n)$ from one another w.h.p.*

Proof. W.l.o.g, we look at the reverse walk starting from s. Define the strict place at which the walk ends to be $s + \sum_{j \in \mathrm{RM}} 2^{-j}$ where $\mathrm{RM} = \{j | j'\text{th move is right}\}$. We now turn to bound the distance the walk ends from its strict

[11] This paper does not address the issue of hotspots, though traditional mechanisms such as multiple replica placement can be adapted to the distributed hash table framework.

place. By goodness property (4) (Claim 4.1), with probability 1/2 in case the walk starts with a left link, one is found within a distance $(3\log n)/n$ from s. Likewise, with probability at least 1/2 a right link is found within a distance $(3\log n)/n$ from $r+1/2$. If the desired link is not found, then again with probability 1/2 we have a link of the same desired level within additional $(3\log n)/n$ distance, and so on. When we finally find a link, we continue the series of experiments looking for the link dictated by the next move, and so on. This geometric experiment continues up to $\log n$ levels. Therefore with probability $1-\left(\frac{1}{2}\right)^{2\log n} = 1-1/n^2$ the last link in the walk is within $(3\log n)/n)(2\log n) = (6\log^2 n)/n$ distance from the strict distance determined by the walk. Therefore we get that the two origins of the path, r_1 and r_2, satisfy $d(r_1, r_2) \le (12\log^2 n)/n$. □

Finally, the connectivity properties of the Viceroy are as follows:

THEOREM 6.6. *If n servers are present in a Viceroy network, then the out-degree of each node is 7 (only 5 of which are used in the simple version), the expected indegree is $O(1)$ and the largest in-degree is $O(\log n)$ w.h.p. (over the identifier and level choice.)*

PROOF. Fix any node s. Then the only nodes that may point to s are those of level $s.level$ and $s.level-1$ in $stretch(s, \text{PREVONLEVEL}(s))$ and $stretch(s, \text{PREVONLEVEL}(s-1/2^{s.level}))$. The result then immediately follows from goodness property (5) (Claim 4.1). □

7. IMPROVED LOOKUP

The simple routing above achieves good expected behavior, but does not yield the desired $O(\log n)$ dilation with high probability. In this section we show how to extend the lookup subroutine to achieve this goal.

The main problem resides in the third phase of the lookup algorithm where we must traverse the ring, even though there might be $O(\log^2 n)$ servers between the current server and the target. An alternative approach that we now describe maintains the decentralized spirit of the Viceroy network and allows us to leave the basic construction unaltered. The idea is that the combination of the global and level rings can yield a better local routing mechanism than the global ring alone.

DEFINITION 7.1. *The* hopping path *from s to t is the one obtained by starting with $cur = s$ and setting $cur = cur.nextonlevel$ if $cur.nextonlevel \in stretch(cur, t)$, $cur = cur.prevonlevel$ if $cur.prevonlevel \in stretch(cur, t)$, otherwise setting $cur = cur.successor$ or $cur = cur.predecessor$ so as to minimize the distance to t. The process is repeated until the* clockwise-closest *node to t is found. The* hopping length *from s to t, denoted $hops(s, t)$, is the number of servers in the hopping path.*

We will show that hopping allows one to route through a poly-logarithmic number of nodes in a logarithmic number of steps w.h.p. By using this routing scheme in the third phase of our lookup algorithm, we will thus achieve the stronger bound we aim for. We modify the third phase lookup algorithm as in Figure 5.

LOOKUP(x, y): (third phase)

Traverse ring: If cur is the clockwise-closest server to x then compute a response to the lookup. Else, if $cur.nextonlevel \in stretch(cur, x)$ then $cur = cur.nextonlevel$, if $cur.prevonlevel \in stretch(cur, x)$ then $cur = cur.prevonlevel$, else set $cur = cur.successor$ or $cur = cur.predecessor$, whichever is closer to x. Repeat phase.

Figure 5: Modified third phase of the LOOKUP algorithm.

7.1 Improved lookup analysis

LEMMA 7.2. *Let us consider a stretch of $O(\log^2 n)$ servers between s and t in a sane and good network. With high probability, the hopping length between s and t satisfies $hops(s, t) = O(\log n)$.*

PROOF. Consider a hopping path on the stretch. It will appear as a sequence of hops on the same level until that level has no more nodes within the stretch, then another sequence of hops on another level, and so on. For the remainder, we will only consider the hopping path before it gets within $c\log n$ single-steps of the target, and show that this length is $O(\log n)$ with high probability. The final $c\log n$ single-steps can take at most $O(\log n)$ hops. We'll call the first set of hops the "hopping phase" and the second set the "stepping phase".

The first task is to bound the number of times the level changes during the hopping phase. Notice that only one level that occurs in the stepping phase can be part of the hopping phase. Let's assume that the levels of the servers in the stepping phase are chosen sequentially until $\log n/4$ distinct levels have been selected, and let X denote the number of trials necessary for this to occur. Because at any point fewer than $\log n/4$ levels have been selected, then the probability of selecting a new level at any point is at least 1/2. The expected value for X is clearly $2\log n/4$. Using a Chernoff bound we have that

$$\Pr[X > c\log n] \le e^{(c-1/2)^2 \log n/6} = \frac{1}{n^{(c-1/2)^2/6}} .$$

Thus with high probability, at least $\log n/4$ levels are in the stepping phase and thus are not in the hopping phase.

Now consider P_i to be the event that the ith server is on the hopping path. We'll use the principle of deferred decisions, and assume that the levels are selected as the hopping path proceeds from s to t, removing a candidate level each time a level change occurs. Besides the nodes that are in the path because a level change occurs (there can be at most $(3\log n) - (\log n)/4$ of these), P_j occurs only if it selects the same level as the last P_i to occur. The selection will be uniform out of the levels that haven't been in the hopping phase thus far. Using the bound we have on the number of different levels of hopping, the probability that P_i occurs is at most $4/\log n$.

We can bound the number, P, of successful P_i events of this type using the Chernoff bounds. In a stretch of $d\log^2 n$ servers, then have

$$\Pr[P > 8d\log n] < e^{-(4d\log n)/3} < \frac{1}{n^d}.$$

Thus with high probability, there are $O(\log n)$ level changes, $O(\log n)$ "continuation" steps of the above type, and $O(\log n)$ steps in the stepping phase. ∎

The analysis of the dilation of the full Viceroy network differs from the simple one only in the last phase. Hence, we obtain the following as an immediate result.

THEOREM 7.3. *If n servers are present in a Viceroy network, then the (worst-case) dilation of the network is $O(\log n)$ w.h.p. (over the identifier and level choice.)*

We note that hopping does not harm the load incurred during the third phase of the lookup because it only *reduces* the chance of a server being in the hopping path. However, the load from the first two phases leaves the expected load as before.

7.2 Bounding Indegrees - The Bucket Solution

So far we have concentrated on fixing the outdegree of nodes and creating a sparse network. This obviously yields small average indegree. However, the largest indegree in the network might still be as large as $\log n$ and thus a single server leaving would cause $\log n$ linkage changes. To combat this we add another background process to the system which we call "buckets." Unfortunately, the details of the bucket mechanism are more involved than the rest of Viceroy so we only sketch its operation here.

Our approach is to improve the identifier and level selection procedures so that in stretches of length $(\log n)/n$ we will have a constant number of servers (between c_1 and c_2 for some fixed c_1 and c_2) from each "sane" level. We achieve this by maintaining a distributed coordination mechanism between contiguous, non-overlapping sets of servers on the ring, *buckets*, consisting of $O(\log n)$ servers each. Inside each bucket, we maintain a simple ring (which mostly overlaps the general ring, except that the end points of the bucket are also connected). The buckets are maintained so that several properties hold:

Size: The size of a bucket is always $\Theta(\log n)$, that is whenever the size of a bucket drops below $\log n$ it merges with a neighbor bucket, and whenever the size exceeds $c \log n$ the bucket splits into two. Such merging and splitting might involve reassignment of levels to *all* members of the bucket. Within the bucket, a precise estimate of $\log n$ is maintained.

Diversity: The levels inside a bucket are *not* assigned at random, rather it is assured that from each level in $[1..\log n]$ there is at least one member and no more than c members. This assures that the indegree of any server is bounded by $2c$. To maintain this last property it might be necessary to reassign a new level to some remaining server in the bucket when a server leaves the system (and the bucket).

Overhead: Note that when a node leaves this can cause at most $O(1)$ other nodes to change their level selection. Also a join or leave can cause a bucket once in $(c-1)\log n$ steps to reshuffle (which involves $O(\log n)$ changes, so the amortized complexity is $O(1)$.

Finally, buckets are also natural units for maintaining replicated data, e.g., on routing information, for fault tolerance.

Acknowledgments

We thank Udi Wieder for a careful reading of the paper and John Mitchell for useful discussions.

8. REFERENCES

[1] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. "Linear Hashing". Journal of the ACM Vol. 46, No. 5, 1999, pp. 667–683.

[2] L. Barriére, P. Fraigniaud, E. Kranakis and D. Krizanc. "Efficient routing in networks with long range contacts". *15th International Symposium on Distributed Computing (DISC '01)*, Lecture Notes in Computer Science vol. 2180, Springer, 2001 pp. 270–284.

[3] A. Fiat and J. Saia. "Censorship resistant peer-to-peer content addressable networks". Proceedings of the 13th ACM-SIAM Symp. on Discrete Algorithms, 2002.

[4] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. "Scalable, distributed data structures for Internet service construction. *Proc. Fourth Symposium on Operating System Design and Implementation (OSDI 2000)*, October 2000.

[5] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 654–663, May 1997.

[6] J. Kleinberg. "The small world phenomenon: An algorithmic perspective". Proceedings of the 32nd ACM Symposium on Theory of Computing, May 2000, pp. 163–170. (A shorter version available as "Navigation in a Small World", *Nature* 406, August 2000, pp. 845.)

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. "OceanStore: An Architecture for Global-Scale Persistent Storage", Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[8] N. Lynch, D. Malkhi and D. Ratajczak. "Atomic Data Access in Distributed Hash Tables", Proceedings of the International Peer-to-Peer Symposium, March 2002.

[9] T. Lai and D. Wood. "Adaptive heuristics for binary search trees and constant linkage cost". *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pp. 72-77, San Francisco, CA, 1991.

[10] M.-J. Lin, K. Marzullo and S. Masini, "Gossip versus Deterministically Constrained Flooding on Small Networks", *Proceedings of the 14th International Conference on Distributed Computing*, 2000, pp. 253–267.

[11] R. J. Lipton and J. F. Naughton. "Clocked adversaries for hashing". *Algorithmica*, Vol. 9(3), 1993, pp. 239–252.

[12] W. Litwin, M.A. Neimat, D. A. Schneider. "LH*-A scalable, distributed data structure". *ACM*

Transactions on Database Systems, Vol. 21(4), 1996, pp. 480–525.

[13] R. Motwani and P. Raghavan. "**Randomized Algorithms**". Cambridge University Press, 1995.

[14] T. Ottmann and D. Wood. "Updating binary trees with constant linkage cost". *International Journal of Foundations of Computer Science*, 3, 1992, pp. 479–501.

[15] G. Pandurangan, P. Raghavan and E. Upfal. "Building low-diameter p2p networks". *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, 2001.

[16] C. Plaxton, R. Rajaram, and A. Richa. "Accessing nearby copies of replicated objects in a distributed environment". *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA 97), pp. 311–320, June 1997.

[17] W. Pugh. "Skip Lists: A probabilistic alternative to balanced trees". *Communications of the ACM*, vol 33(6), pp. 668–676, 1990.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. "A scalable content-addressable network". *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. August 2001.

[19] H. J. Siegel. "Interconnection networks for SIMD machines". *Computer* 12(6):57–65, 1979.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications". *Proceedings of the SIGCOMM 2001*, August 2001.

[21] B. Y. Zhao, J. D. Kubiatowicz and A. D. Joseph. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". U. C. Berkeley Technical Report UCB/CSD-01-1141, April, 2001.