On Name Resolution in Peer-to-Peer Networks *

Xiaozhou Li C. Greg Plaxton

November 2002

Abstract

An efficient name resolution scheme is the cornerstone of any peer-to-peer network. The foundation of an efficient name resolution scheme is a dynamic network topology that determines the neighbor relationships to be maintained by the nodes in the network. The name resolution scheme proposed by Plaxton, Rajaraman, and Richa, which we hereafter refer to as the PRR scheme, is a scalable scheme that also provides provable locality properties on a certain class of growth-restricted metric spaces. On arbitrary metric spaces, however, some performance bounds of PRR are significantly weakened. In this paper, we define a class of network topologies called *hyperdelta networks* and observe that the PRR topology may be viewed as a random hyperdelta network. We then propose SPRR (simplified PRR), a variant of the PRR scheme that performs well on arbitrary metric spaces. SPRR imposes additional constraints on PRR neighbor selection by placing the nodes on a cycle. Although SPRR does not provide as strong locality properties as PRR, it exploits locality heuristically yet effectively. Finally, a significant level of fault tolerance can be achieved in SPRR without adding much complexity.

*Department of Computer Science, University of Texas at Austin, Austin, TX 78712. This research was supported by NSF Grant CCR–9821053. Email: {*xli, plaxton*}@*cs.utexas.edu*.

1 Introduction

An efficient name resolution scheme is the cornerstone of any peer-to-peer network. Given a name, the task of name resolution is to determine the value to which the name maps. An important use of name resolution is to find the location of a data item. At a minimum, a name resolution scheme supports the following operations: (1) name operations: lookup (looking up a name), insert (inserting a name), and delete (deleting a name); (2) network operations: join (adding a node to the network) and leave (removing a node from the network). Since a peer-to-peer network can have a large number of nodes, scalability (i.e., efficient support for the above operations), is a fundamental requirement for name resolution. Furthermore, locality (i.e., the distance traveled by each hop), is not to be ignored. For example, a 10-hop path in a global peer-to-peer network in which each hop is intercontinental is likely to be dramatically inferior to a 10-hop path in which most or all of the hops are "local" (e.g., within a single college campus). Finally, a name resolution scheme should have strong fault tolerance properties.

Considerable research effort has been invested in the design of efficient name resolution schemes and several schemes have been proposed, including the PRR scheme proposed by Plaxton *et al.* [20], Chord [27], CAN [23], and Viceroy [16]. The routing method of PRR, which is a variant of hypercubic routing, is later also used by Tapestry [29], Pastry [25], and Kademlia [17].

The foundation of an efficient name resolution scheme is a dynamic network topology that determines the neighbor relationships to be maintained by the nodes in the network. For the sake of brevity, we identify a name resolution scheme with its dynamic network topology in this paper. For example, PRR maintains a logarithmic number of neighbor pointers at each node, and each neighbor of a node u is the nearest node to u with an ID matching that of u in a certain number of bit positions. On arbitrary metric spaces, PRR guarantees logarithmic number of hops between any two nodes. On a certain class of growth-restricted metric spaces, PRR provides provable locality properties and efficient support for join and leave operations. Maintaining the PRR neighbor pointers, however, is a nontrivial task, especially if the distance function is changing, or if nodes are frequently joining or leaving the network. Although recent research results have reduced the restriction on the metric spaces [11, 12], providing similar locality properties on general metric spaces remains an open problem.

When applied to arbitrary metric spaces, some performance bounds established by PRR are considerably weakened. For example, on growth-restricted metric spaces, the in-degree of a node is $O(\log n)$ expected and $O(\log^2 n)$ with high probability (whp). (We say that an event happens *with high probability* or *whp* if it fails to occur with probability at most n^{-c}, where n is the number of nodes in the network and c is a positive constant that can be set arbitrarily large by adjusting other constants in the relevant context.) However, consider a star network with n nodes such that one (special) node is distance 1 from all other (non-special) nodes and the distance between any two non-special nodes is 2. Therefore, the special node is a neighbor of every non-special node, regardless of the values of the individual IDs. This is undesirable because if the special node leaves the network, all other nodes have to update their neighbor tables. In addition, the special node carries a disproportionate routing workload because it is a neighbor of many other nodes.

In this paper, we define a class of network topologies called *hyperdelta networks* and observe that the PRR topology may be viewed as a random hyperdelta network. We then propose SPRR (simplified PRR), a simple variant of the PRR scheme that performs well on arbitrary metric spaces. Compared to Chord, SPRR has matching or improved high probability time bounds and matching or better expected running times for all name resolution operations. For example, the join operation in SPRR runs in $O(\lg n)$ time with high probability (whp), whereas, given the Chord definition of fingers, the join operation in Chord requires $\Omega(\lg^2 n)$ time to succeed whp. (The reason for this is that for any $\varepsilon > 0$, there is a $n^{-\varepsilon}$ probability that $\Omega(\lg^2 n)$ nodes will need to update their fingers as a result of the join.) In a dynamic environment, the ability to quickly add or remove a node from the network is particularly important.

We then show that SPRR exploits locality effectively. We argue that SPRR is likely to have good locality properties on any metric space, and we give the rigorous locality properties on the ring metric. Although giving up provable locality properties for simplicity may affect performance, we argue that the loss of performance is likely to be minor, while simplicity not only is a desirable feature in system design, but also helps reasoning about the correctness of a scheme, a point which we will elaborate on in Section 6.

Finally, we show that fault tolerance can be achieved in SPRR without adding much complexity. We propose a novel name replication strategy ensuring names be looked up whp in a random fault model where each node has a constant probability of being down. We show that lookups remain efficient (i.e., taking logarithmic number of hops) in this random fault model.

The rest of the paper is organized as follows. Section 2 presents SPRR on uniform metric space. Section 3 shows how SPRR exploits locality. Section 4 shows how fault tolerance can be achieved in SPRR. Section 5 discusses related work. Section 6 discusses future work. Section 7 concludes the paper.

2 Fault-Free Name Resolution on Uniform Metric Space

In this section, we define hyperdelta networks and present a simplified version of SPRR that is suitable for a basic network model in which nodes are fault-free and the distance between any pair of nodes is equal (uniform metric space). We show that this version of SPRR not only greatly simplifies PRR, but also retains the scalability properties of PRR. Of course, for real peer-to-peer applications, locality and fault tolerance have to be taken into account. We address locality and fault tolerance issues in Sections 3 and 4, respectively.

We first introduce a few definitions and notations that will be used throughout the paper. Every node and every object has a random binary string as its identifier (ID). Object IDs are also called *names* in this paper. For the purpose of analysis, we view an ID as an infinite sequence of random bits, and we assume that no two IDs are identical. In practice, the length of an ID is chosen to be long enough (say, 128 bits) so that the chance of having two identical IDs is negligible. ID bits are numbered starting from 0. At times, we identify a node with its ID when no confusion could arise. Let V be the set of nodes in the network; $n = |V|$ be the number of nodes in the network; $x[i]$ be bit i of ID x; $match(x, y)$ be the length of the longest common prefix shared by ID x and ID y; $N(\beta)$ be the set of nodes prefixed by bit string β; $F(u, i)$ be the set $\{v \in V : match(u, v) = i\}$; $\Gamma(u, i)$ be the set $\{v \in V : match(u, v) \geq i\}$. Note that $u \in \Gamma(u, i)$, for all i. The number of neighbors of a node is called the *out-degree* of the node; the number of nodes of which the node is a neighbor is called the *in-degree* of the node.

The rest of this section is organized as follows. Section 2.1 defines hyperdelta networks. Section 2.2 presents the basic organization of SPRR. Section 2.3 presents the basic name resolution operations in SPRR. Section 2.4 discusses several alternative implementations of SPRR. Section 2.5 analyzes the properties of SPRR.

2.1 Hyperdelta Networks

We begin by reviewing a few standard static network topologies. (See Leighton [14] for a thorough coverage of these topologies.) Delta networks are a class of n-input n-output switching networks that have a unique path from any input to any output, and that the label sequence of every path to the same output is the same. A butterfly is an example of a delta network. A hypercube is a topology closely related to a butterfly: a hypercube can be viewed as a "collapsed" version of a butterfly in which each of the n columns in the butterfly corresponds to a single node in the hypercube. The same collapsing operation can be applied to any delta network to obtain a hypercube-like network. We call such a collapsed delta network a *hyperdelta network*. Although hyperdelta networks have desirable properties in the static setting (e.g., logarithmic degree and diameter), in a dynamic setting, where nodes keep joining and leaving the network, it is no

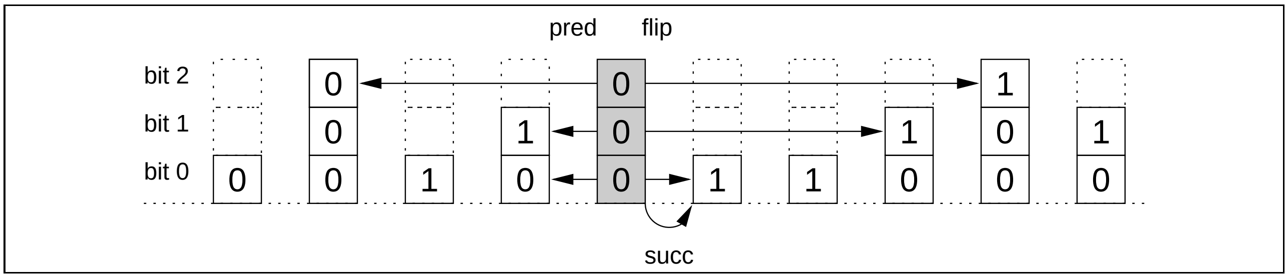

Figure 1: An example of SPRR.

longer feasible to assign fixed IDs to nodes and follow the strict neighbor rules as in the static setting. To solve this problem, PRR has every node choose its own ID at random, and the bit-i neighbor of a node u is the nearest node in $F(u, i)$. Therefore, PRR can be viewed as a minimum-cost random hyperdelta network. As discussed in Section 1, although PRR provides strong locality properties on a certain class of growth-restricted metric spaces, some of its performance bounds (e.g., in-degree) are considerably weakened on arbitrary metric spaces. To overcome this weakness, we now propose SPRR, a variant of the PRR scheme that performs well on arbitrary metric spaces.

2.2 Basic Organization of SPRR

SPRR first places all the nodes on an arbitrary *logical ring* which is independent of node IDs, and requires that nodes choose their neighbors conforming to the logical ring. To be precise, every node u maintains the following three types of neighbors:

- *Flip neighbors.* The bit-i *flip neighbor* of u, denoted by $u.\mathit{flip}[i]$, is the first node v clockwise from u such that $v \in F(u, i)$. If $F(u, i) = \emptyset$, then $u.\mathit{flip}[i] = \mathbf{nil}$.
- *Predecessors.* The bit-i *predecessor* of u, denoted by $u.\mathit{pred}[i]$, is the first node v counterclockwise from u such that $v \in \Gamma(u, i)$. If u is the only node in $\Gamma(u, i)$, then $u.\mathit{pred}[i] = u$.
- *Successor.* The bit-0 successor, or simply *successor*, of u, denoted by $u.\mathit{succ}$, is the "opposite" of bit-0 predecessor (i.e., $u.\mathit{pred}[0] = v$ iff $v.\mathit{succ} = u$). Note that we only maintain one successor pointer at each node.

At a high level, the SPRR topology may be termed random cyclic hyperdelta network. Apparently, imposing additional constraints on neighbor selection increases the average edge length. Bounding this increase is still an open problem and is our future work (see Section 6). Figure 1 shows an example of an SPRR topology. For simplicity, we only show the neighbor pointers of a single node at the first three bit positions.

SPRR admits a simple recursive definition: an SPRR network is composed of two SPRR networks, one consisting of the nodes that begin with 0 and the other the nodes that begin with 1. Such a recursive definition is a hallmark of hypercubic networks. We believe that this recursive structure will prove crucial in our future work (see Section 6).

We next discuss how names are handled in SPRR. When a name is inserted, it is stored at a certain node, called the *handler* of the name, which is responsible for resolving the name. Each node u maintains a local name database, denoted by $u.db$, to store the names for which it is responsible. Handlers are assigned as follows. Let the *best match set* of a name a, denoted by $B(a)$, be the set

$$\{u \in V : \forall v \in V, \mathit{match}(u, a) \geq \mathit{match}(v, a)\}.$$

We call $match(u, a)$, where $u \in B(a)$, the *depth* of the best match set. When a name a is inserted, the insert request is forwarded until a node in $B(a)$ is reached. This node is designated as the handler of the name. When a name is later looked up or deleted, additional work has to be done to locate the handler of the name because there may be multiple nodes in the best match set. We discuss the details of each operation in the next section. Note that all the nodes in a best match set is organized in a cycle by the predecessor pointers at a certain bit position.

Although we do not provide rigorous load balance analysis of SPRR in this technical report, it is clear from the above description that SPRR achieves load balance properties similar to those established by Chord. That is, the expected load of every node is the same, although the load of the heaviest node is $O(\lg n)$ of that of the lightest node. While Chord uses "virtual nodes" (i.e., every physical node simulates $O(\lg n)$ logical nodes) to smooth out the imbalance so that the load of every node is within a constant factor of each other whp, we use name replication to achieve the same goal. (In fact, name replication achieves both load balance and fault tolerance.) We discuss our name replication strategy in Section 3.1.

2.3 Basic Operations

With the above definitions, we are now ready to present the basic operations of name resolution. Central to the name operations (lookup, insert, and delete) is finding the handler of a name. The process is divided into two phases: (1) the *bit-correcting* phase, during which a node in the best match set is reached; (2) the *walking* phase, during which the nodes in the best match set are traversed by following the predecessor pointers until the handler of the name is found. For an insert operation, only the bit-correcting phase is needed, while for a lookup or delete operation, both phases are needed. For network operations (join and leave), existing nodes need to be informed about the arrival or departure of a node. The predecessor pointers and the successor pointer enable SPRR to support join and leave efficiently.

We next describe each operation in detail and present our code for these operations. Our code is written in a style similar to Gouda's Abstract Protocol Notation [9], with some minor variations. In our code, keyword $\mathbf{you}$ is only defined when a message is being received and denotes the sender of the message, and keyword $\mathbf{me}$ denotes the current node.

- *Lookup.* To look up a name, a node needs to find the handler of the name. The process of finding the handler includes both the bit-correcting phase and the walking phase. During the bit-correcting phase, a node forwards the lookup request to one of its flip neighbors, which matches the name in more bits, until the lookup request cannot be forwarded further, in which case the current node belongs to the best match set. The walking phase is needed because the first node reached in the best match set may not be the handler of the name. Thus, we need to traverse the best match set by following the predecessor pointers until we locate the handler. The code for lookup is shown in Figure 2.

- *Insert.* When a name is inserted, we only need to find an arbitrary node in the best match set and hence only the bit-correcting phase is needed. When the lookup request cannot be forwarded any further through flip neighbors, the current node belongs to the best match set and is the handler of the name.

- *Delete.* Deleting a name is largely the same as looking up a name, except that when a name is located, it is deleted from the local name database, instead of being looked up. We omit the code for insert and delete, because it is largely the same as that for lookup.

- *Join.* When a new node joins, the new node first finds an arbitrary node in the network through some external mechanism. We assume that some external mechanism enables the new node to find such a node, an assumption made by many other schemes. As will be discussed in Section 3.3, to exploit

```
□ true → {lookup}
    a := arbitrary name;
    send jump(me, a, 0) to me;

□ rcv jump(x, a, i) →
    do (id[i] = a[i]) → i := i + 1;
    od;
    if (flip[i] ≠ nil) → send jump(x, a, i + 1) to flip[i];
    □ (flip[i] = nil) → send walk(x, me, a, i) to me;
    fi;

□ rcv walk(x, y, a, i) →
    if (a ∉ db ∧ pred[i] ≠ y) → send walk(x, y, a, i) to pred[i];
    □ (a ∈ db ∨ pred[i] = y) → send reply(a, db.find(a)) to x;
    fi;

□ rcv reply(a, value) →
    skip;
```

Figure 2: Code for lookup.

locality, it is desirable for distances on the logical ring to be correlated with actual distances in the metric space. Thus it is desirable for the external mechanism to determine a node close to the new node. The new node then sets its successor to be that node (i.e., it places itself on the logical ring immediately counterclockwise to that node) and generates its own ID. The new node then builds its own neighbor table by consulting a sequence of nodes, starting from its successor. During this process, the neighbor pointers of other nodes are also updated. The code for join is shown in Figure 3. The code used by both join and leave is shown in Figure 4.

- *Leave.* When a node leaves, it needs to inform other nodes to update their neighbor pointers, a process similar to join. The code for leave, which is largely symmetric to that for join, is shown in Figure 5.

We remark that although we content ourselves with the compactness and clarity of our code, certain optimizations are possible in actual implementation. For example, messages sent to a node itself can be replaced by function calls. Therefore, these messages are not counted in our analysis.

2.4 Alternative Implementations

Besides the implementation presented above, SPRR admits several alternative implementations (e.g., the one presented in our recent workshop paper [15]). Below we mention a few other possibilities.

1. In the implementation presented above, if we are willing to expend a few messages to locate the successor, then the successor pointer need not be maintained, because it can be located using only $flip[0]$ and $pred[0]$ pointers.
2. The most straightforward implementation perhaps is to maintain predecessors and successors, but not flip neighbors, at all levels. Hence, nodes are organized into various doubly-linked cycles. Flip neighbors are not needed because they can be located by following the successor pointers. Although this implementation satisfies all the performance bounds stated in the rest of this paper, the constant

```
□ true → {join}
    succ := arbitrary node in the network, nil if the network is empty;
    join_level(succ, 0);

□ rcv arrive(i, b) →
    x, pred[i] := pred[i], you;
    if (id[i] ≠ b ∧ flip[i] = nil) → flip[i] := you;
    □ (id[i] = b ∨ flip[i] ≠ nil) → skip;
    fi;
    if (id[i] = b) → y, z := flip[i], me;
    □ (id[i] ≠ b) → y, z := me, flip[i];
    fi;
    send ack_arrive(x, y, z, i) to you;

□ rcv ack_arrive(x, y, z, i) →
    pred[i], flip[i] := x, y;
    send update_flip(me, i, id[i]) to pred[i];
    join_level(z, i + 1);

macro join_level(x, i)
    if (x ≠ me ∧ x ≠ nil) → send arrive(i, id[i]) to x;
    □ (x = me ∨ x = nil) → send update_succ(me) to pred[0];
    fi;
```

Figure 3: Code for join.

factor in the expected lookup time is doubled. Since lookup is the most important operation, we opt for an implementation with lower constant factors.

3. Yet another implementation is to maintain predecessors, successors, and flip neighbors at all levels. The disadvantage of this implementation, however, is higher node degree.

Therefore, in the balance of this paper, we assume the implementation presented in Section 2.2.

2.5 Analysis

In this section, we analyze the properties of SPRR. Our main result is that all operations take $O(\lg n)$ constant-size messages (or application-level hops) whp. For the two network operations join and leave, this

```
□ rcv update_flip(x, i, b) →
    if (id[i] ≠ b ∧ flip[i] ≠ x) → flip[i] := x; send update_flip(x, i, b) to pred[i];
    □ (id[i] = b ∨ flip[i] = x) → skip;
    fi;

□ rcv update_succ(x) →
    succ := x;
```

Figure 4: Common code used by both join and leave.

```
□ true → {leave}
    leave_level(succ, 0);

□ rcv depart(x, i, b) →
    pred[i] := x;
    if (flip[i] = you) → flip[i] := nil;
    □ (flip[i] ≠ you) → skip;
    fi;
    if (id[i] = b) → y := me;
    □ (id[i] ≠ b) → y := flip[i];
    fi;
    send ack_depart(y, i) to you;

□ rcv ack_depart(x, i) →
    send update_flip(x, i, id[i]) to pred[i];
    leave_level(x, i + 1);

macro leave_level(x, i)
    if (x ≠ me ∧ x ≠ nil) → send depart(pred[i], i, id[i]) to x;
    □ (x = me ∨ x = nil) → send update_succ(succ) to pred[0];
    fi;
```

Figure 5: Code for leave.

represents a significant improvement over the $O(\lg^2 n)$ message bound established by Chord. With respect to the name operations, SPRR matches Chord in terms of both expected and whp bounds.

We now present a series of lemmas and theorems. One important observation is that given any node u and bit i, each of the remaining nodes v independently has a probability of exactly 2^{-i-1} of belonging to $F(u, i)$. This observation allows us to use Chernoff bounds arguments to establish several of the claims below.

Lemma 2.1 *Whp,* $|N(\beta)| = \Theta(\lg n)$*, where* β *is an arbitrary bit string of length* $\lg n - \lg\lg n - c$*, for some sufficiently large constant* c*.*

Proof: Clearly, $\mathrm{E}\,[|N(\beta)|] = 2^c \lg n$. Chernoff bounds imply that $|N(\beta)|$ lies within a constant factor of its expectation whp. Thus, $|N(\beta)| = \Theta(\lg n)$ whp. ■

Lemma 2.2 *Both the bit-correcting and the walking phases take* $O(\lg n)$ *hops whp.*

Proof: By Lemma 2.1, when we look up a name a, then within $\lg n - \lg\lg n - c$ bit-correcting hops, the lookup request reaches a node in $N(\beta)$, where β is a bit string of length $\lg n - \lg\lg n - c$, for some sufficiently large constant c. Subsequent hops only visit the nodes in $N(\beta)$ and $|N(\beta)| = \Theta(\lg n)$ whp. Thus, both the bit-correcting and walking phases take $O(\lg n)$ hops whp. ■

Theorem 1 *All name operations take* $O(\lg n)$ *hops whp.*

Proof: Immediate from Lemma 2.2. ■

Lemma 2.3 *The expected depth of the best match set is* $\lg n + O(1)$.

Proof: Let X denote the depth of the best match set. Then

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i\geq 1} \Pr[X = i] \cdot i \\
&= \sum_{i\geq 1} \Pr[X \geq i] \\
&= \sum_{i\geq 1} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&= \sum_{i=1}^{\lg n - 1} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) + \sum_{i\geq \lg n} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&\leq \lg n - 1 + \sum_{i\geq \lg n} \left(1 - \left(1 - \frac{1}{2^i}\right)^n\right) \\
&= \lg n - 1 + \sum_{i\geq 0} \left(1 - \left(1 - \frac{1}{2^i n}\right)^n\right)
\end{aligned}$$

We observe

$$\begin{aligned}
\left(1 - \frac{1}{2^i n}\right)^n &= \left(\left(1 - \frac{1}{2^i n}\right)^{2^i n - 1} \cdot \left(1 - \frac{1}{2^i n}\right)\right)^{1/2^i} \\
&\geq \left(\frac{1}{e}\left(1 - \frac{1}{2^i n}\right)\right)^{1/2^i} \\
&\geq \left(\frac{1}{2e}\right)^{1/2^i}.
\end{aligned}$$

The first inequality in the above derivation holds because

$$\left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e} \leq \left(1 - \frac{1}{n}\right)^{n-1},$$

which is implied by the following more general inequality:

$$\left(1 + \frac{x}{p}\right)^p \leq \frac{1}{e} \leq \left(1 + \frac{x}{p}\right)^{p + x/2}, \text{ for all positive reals } x \text{ and } p.$$

For a proof of this inequality, see, e.g., [18]. Thus,

$$\begin{aligned}
\mathrm{E}[X] &\leq (\lg n - 1) + \sum_{i\geq 0} \left(1 - \left(\frac{1}{2e}\right)^{1/2^i}\right) \\
&\leq \lg n + O(1).
\end{aligned}$$

The series $\sum_{i\geq 0} \left(1 - \left(\frac{1}{2e}\right)^{1/2^i}\right)$ is bounded by a constant because the ratio between successive terms is $1 + \left(\frac{1}{2e}\right)^{1/2^i}$, which is at least $1 + \frac{1}{2e}$. ∎

Lemma 2.4 *The expected size of the best match set is constant.*

Proof: Without loss of generality, assume the name a to be looked up is all 0's. Let X be the size of the best match set and let n_j be the number of nodes prefixed by j 0's. Consider the maximum k such that $n_k \geq i$. In order for $X = i$, it is necessary that $n_k = i$ and $n_{k+1} = 0$. Note that $\Pr[n_{k+1} = 0 \mid n_k = i] = \frac{1}{2^i - 1}$. Thus $\Pr[X = i] = O(2^{-i})$, and hence $\mathrm{E}[X] = O(1)$. ∎

Theorem 2 *The expected number of messages needed by a name operation is* $\frac{1}{2}\lg n + O(1)$.

Proof: The expected number of messages needed in the bit-correcting phase is half of the depth of the best match set. The expected number of messages needed in the walking phase is bounded by the size of the best match set. By linearity of expectation and Lemmas 2.3 and 2.4, the number of messages needed by a name operation is $\frac{1}{2}\lg n + O(1)$. ∎

Lemma 2.5 *Every node has at most* $\lg n + O(\sqrt{\lg n})$ *flip neighbors whp.*

Proof: Let u be the node under consideration. Starting from bit 0, we divide the ID of u into three segments A, B, and C, such that the lengths of A and B are $\lg n$ and $c \lg n$, respectively, where c is a sufficiently large constant. Let X_A, X_B, and X_C be the number of flip neighbors in segments A, B, and C, respectively. Clearly, $X_A \leq \lg n$ at all times. To bound X_C, we first define sets G_i, for all $i \geq 0$, as

$$G_i = \{v \in V : v \neq u \wedge match(u, v) \geq i\}.$$

Then for any node v, independently, $\Pr[v \in G_i] = \frac{1}{2^i}$ and thus, $\mathrm{E}[|G_i|] = \frac{n-1}{2^i} \leq \frac{n}{2^i}$. We observe that $X_C \leq |G_{(c+1)\lg n}|$. Thus,

$$\begin{aligned}
\mathrm{E}[X_C] &\leq \mathrm{E}\left[|G_{(c+1)\lg n}|\right] \\
&\leq \frac{n}{2^{(c+1)\lg n}} \\
&= n^{-c}.
\end{aligned}$$

Markov's inequality implies $\Pr[X_C \geq 1] \leq \mathrm{E}[X_C] = n^{-c}$, that is, $X_C = 0$ whp. To bound X_B, we first observe that for any node v, independently,

$$\Pr[v \in F(u, i)] = \frac{1}{2^{i+1}}$$

and $\mathrm{E}[|F(u, i)|] = \frac{n-1}{2^{i+1}} \leq \frac{n}{2^{i+1}}$. Again, by Markov's inequality, we have

$$\begin{aligned}
\Pr[F(u, i) \neq \emptyset] &= \Pr[|F(u, i)| \geq 1] \\
&= \mathrm{E}[F(u, i)] \\
&\leq \frac{n}{2^{i+1}}.
\end{aligned}$$

Thus,

$$\Pr\left[X_B \geq c'\sqrt{\lg n}\right] \leq \binom{c \lg n}{c'\sqrt{\lg n}} \Pr\left[\bigwedge_{i=\lg n}^{\lg n + c'\sqrt{\lg n} - 1} F(u, i) \neq \emptyset\right]$$

$$
\begin{aligned}
&\leq \binom{c\lg n}{c'\sqrt{\lg n}} \prod_{i=\lg n}^{\lg n + c'\sqrt{\lg n}-1} \Pr\left[F(u,i) \neq \emptyset\right] \\
&\leq (c\lg n)^{c'\sqrt{\lg n}} \prod_{i=\lg n}^{\lg n + c'\sqrt{\lg n}-1} \Pr\left[F(u,i) \neq \emptyset\right] \\
&= n^{o(1)} \cdot \prod_{i=1}^{c'\sqrt{\lg n}-1} \frac{1}{2^i} \\
&= n^{o(1)} \cdot 2^{(c'\sqrt{\lg n} - c'c'\lg n)/2} \\
&\leq n^{o(1)} \cdot 2^{(c'\lg n - c'c'\lg n)/2} \\
&= n^{o(1)+c'/2-c'c'/2}.
\end{aligned}
$$

The second inequality in the above derivation holds because the dependency between $\Pr\left[F(u,i) \neq \emptyset\right]$ for different i's is in our favor. That is, having a flip neighbor at a certain bit decreases the probability of having a forward neighbor at a different bit.

Thus, $X_A + X_B + X_C \leq \lg n + O(\sqrt{\lg n})$ whp. ∎

Lemma 2.6 *For all sufficiently large positive constants c, for all $i \geq c\lg n$, and for all nodes u, $u.flip[i] =$ $\mathbf{nil}$ whp.*

Proof: For all $i \geq c\lg n$, $\mathrm{E}\left[|F(u,i)|\right] = O(n^{-c})$. The claim of the lemma follows from Markov's inequality and Boole's inequality. ∎

Lemma 2.7 *For all sufficiently large positive constants c, for all $i \geq c\lg n$, and for all nodes u, $u.pred[i] =$ $\mathbf{me}$ whp.*

Proof: Similar to that of Lemma 2.6. ∎

Theorem 3 *The out-degree of every node is $O(\lg n)$ whp.*

Proof: By Lemmas 2.6 and 2.7, we have that every node has $O(\lg n)$ flip neighbors and predecessors whp. Every node also has a single successor. Therefore, the out-degree of every node is $O(\lg n)$ whp. ∎

Theorem 4 *The in-degree of every node is $O(\lg n)$ whp.*

Proof: Fix a node u. Without loss of generality, assume that the ID of u is all 0's. Let the sequence of nodes that precede u on the logical ring, starting from the closest one, be $\langle v_1, v_2, \ldots, v_{n-1}\rangle$. We start with inspecting bit 0 of the IDs of this sequence of nodes. Once we see a 0, we start inspecting bit 1 of those subsequent nodes prefixed by 0, once we see a 0 on bit 1, we start inspecting bit 2 of those subsequent nodes prefixed by 00, and so forth. We keep inspecting until we return to the node u. The key observation is that the nodes inspected in this process are exactly those that have u as one of their flip neighbors. Furthermore, by Lemma 2.6, no node has a flip neighbor at a bit higher than $c\lg n$. Since every node inspected has an independent probability of $1/2$ to increment the index of the bit to be inspected, a Chernoff bound argument implies that the number of nodes inspected can be bounded by $O(\lg n)$ whp. Moreover, Lemma 2.7 implies that the number of nodes that have u as one of their predecessors is $O(\lg n)$ whp. Finally, at most one node has u as its successor. Hence, the in-degree of every node is $O(\lg n)$ whp. ∎

Theorem 5 *A join or leave operation takes $O(\lg n)$ messages whp. The number of existing neighbor table entries that need to be modified is $O(\lg n)$ whp.*

Proof: Immediate from Theorems 3 and 4. ∎

3 Exploiting Locality

In the previous Section, we have shown that SPRR is simple and efficient on the uniform metric space. To be useful in real applications, however, locality has to be taken into account. In this section, we show that locality can be exploited in SPRR without adding much complexity.

The ease of exploiting locality in SPRR comes from the definition of flip neighbors. Unlike Chord, which defines a finger to point to the first node following a certain point on the ID ring, PRR defines a flip neighbor to be the "best" node in a set of candidates. It is exactly this freedom of choice that enables SPRR to exploit locality easily.

Many applications benefit from having multiple copies of a name in the network, for performance or fault tolerance reasons. For example, a name may be replicated to reduce resolution time. SPRR provides locality in the following sense: the expected distance traveled by a lookup decreases as the number of copies increases.

Name replication capability can either be built within a name resolution scheme, or be built as a separate layer. Although the later approach has its own merits (e.g., cleaner interface), we favor the former approach because it provides better lookup performance and better load balance.

Section 3.1 first proposes our name replication strategy; Section 3.2 explains heuristically why SPRR is likely to have good locality property on any metric space; Section 3.3 rigorously proves this locality property on the ring metric.

3.1 Name Replication Strategy

Our name replication strategy is as follows. In the process of inserting a name a, when the insert request reaches a node u in the best match set, u replicates a at r nodes around itself that match a better than the rest of the nodes. This replication strategy can be achieved by simply following the predecessor pointers. We call this set of r nodes, denoted by $R(a)$, the *replication set* of a, In other words, $R(a)$ is a set such that $|R(a)| = r$ and for all $v \in R(a)$ and $w \notin R(a)$, $\mathit{match}(v, a) \geq \mathit{match}(w, a)$. This replication strategy requires a node check its local name database before forwarding lookup request to a neighbor, because now a name can be stored at multiple nodes.

A system can choose an appropriate r for its desired performance or fault tolerance, or choose different r for different names. For example, it can set $r = \Omega(\lg n)$ to ensure that at least one node in the replication set is up whp. If every name is replicated at $\Theta(\lg n)$ nodes using the above replication strategy, then the system is better load balanced, i.e., the load of every node is within a constant factor of each other whp.

3.2 Heuristic Exploitation of Locality

As mentioned above, the ability to exploit locality in SPRR originates from PRR's flexibility to choose a good flip neighbor from a set of candidate nodes. For example, consider the process of choosing the bit 0 flip neighbor of a node u. On average, there are $n/2$ nodes with IDs that differ from the ID of u in bit 0. Among such a large set of nodes, at least one of them is likely to be close to u. Similarly, $\mathrm{E}\left[|F(u, 1)|\right] = n/4$, and so forth. Thus, the number of candidate nodes keeps shrinking with every bit corrected. This implies that the expected distance traveled in order to correct each bit grows with every bit corrected. The speed of growth, of course, depends on the underlying network topology. If the growth is geometric, then the total distance of the hops taken in a lookup operation is dominated by the distance traveled by the last hop in the bit-correcting phase plus the distance traveled in the walking phase. The reason that SPRR has good locality properties is that most of the hops in a lookup operation are bit-correcting hops. Moreover, if a name is replicated at multiple nodes, it is likely to be found before the bit-correcting phase is over.

3.3 The Ring Metric

In this section, we analyze the locality property of SPRR on the ring metric, where the distance between two nodes is the distance between them on a ring, which is also called the *locality ring*.

Although the ring metric is somewhat artificially simple, we remark that it is not totally unrealistic. For example, consider a peer-to-peer network composed of nodes on different universities on different continents. We can arrange the nodes located in the same university in a contiguous region of the ring, and arrange the universities located in the same continent in a bigger nearby region, and so forth.

As discussed in Section 2.2, a node chooses a flip neighbor from a set of candidates by imposing a logical ring on the nodes. Since the logical ring is arbitrary, we can use the locality ring as the logical ring. Employing the replication strategy described in Section 3.1, we establish the following theorem with respect to the ring metric.

Theorem 6 *If a name is replicated at r nodes using the above replication strategy, then the expected distance traveled by a lookup operation is $O(n/r)$.*

Proof: Let a be the name being looked up. Let X denote the size of the best match set. Let d be the distance traveled by the entire lookup operation. Let d_1 be the distance traveled in the bit-correcting phase. Let d_2 be the distance traveled in the walking phase.

By linearity of expectation, $\mathrm{E}[d] = \mathrm{E}[d_1] + \mathrm{E}[d_2]$. To bound $\mathrm{E}[d_2]$, we first observe that if $X \leq r$, then $d_2 = 0$; if $X > r$, then $d_2 \leq n(X - r)$. By Lemma 2.4, we know that $\Pr[X = i] \leq \frac{1}{2^i - 1}$. Thus, we can bound $\mathrm{E}[d_2]$ as follows,

$$\begin{aligned}
\mathrm{E}[d_2] &\leq \sum_{i \geq r+1} n(i-r) \cdot \Pr[X = i] \\
&\leq n \sum_{i \geq r+1} \frac{i-r}{2^i - 1} \\
&= O(n/2^r) \\
&= O(n/r).
\end{aligned}$$

We next bound d_1. Let m to be the smallest integer such that all the nodes that match a in at least m prefix bits are in R, let $R' = \{v : \mathit{match}(v,t) \geq m\}$, and let $Y = |R'|$. We first observe that, in the bit-correcting phase, the lookup operation does not travel beyond the node in R' that is clockwise closest to the originating node. Thus, $\mathrm{E}[d_1]$ is bounded by the average distance between two nodes in R', which is $O(n/Y)$. Thus,

$$\begin{aligned}
\mathrm{E}[d_1] &\leq \sum_{1 \leq i \leq r} \Pr[Y = i] \cdot O(n/i) \\
&= O(n) \cdot \sum_{1 \leq i \leq r} \frac{1}{i} \cdot \Pr[Y = i] \\
&= O(n) \left(\sum_{1 \leq i \leq r/4} \frac{1}{i} \cdot \Pr[Y = i] + \sum_{r/4 < i \leq r} \frac{1}{i} \cdot \Pr[Y = i] \right)
\end{aligned}$$

$$
\begin{aligned}
&= O(n)\left(\sum_{1\le i\le r/4} \Pr\left[Y=i\right] + \frac{4}{r}\cdot \Pr\left[Y > \frac{r}{4}\right]\right) \\
&= O(n)\cdot \Pr\left[Y \le \frac{r}{4}\right] + O(n/r) \\
&\le O(n)\cdot e^{-r/16} + O(n/r) \\
&= O(n/r).
\end{aligned}
$$

The last inequality above is due to an application of Chernoff bounds.

Therefore, $\mathrm{E}\left[d\right] = \mathrm{E}\left[d_1\right] + \mathrm{E}\left[d_2\right] = O(n/r)$. ■

4 Fault-Tolerant SPRR

In the previous section, we have shown how to exploit locality in SPRR. In this section, we show that, without adding much complexity, a significant level of fault tolerance can be achieved.

We adopt a random fault model where every node has a constant probability q of being down. By down, we mean fail-stop faults instead of Byzantine faults. We also assume that a node can detect whether a neighbor is down. With respect to this fault model, our objective is to ensure that fault-tolerant lookup retains the efficiency and locality properties of fault-free lookup.

The rest of this section is organized as follows. Section 4.1 proposes two modifications to the basic construction of SPRR. Section 4.2 describes the fault-tolerant lookup operation. Section 4.3 establishes efficiency and locality properties of the fault-tolerant lookup.

4.1 Modifications to the Basic Construction

Clearly, in a random fault model defined above, a name has to be replicated at $\Omega(\lg n)$ nodes, simply to ensure that at least one node that handles the name is up whp.[1] Furthermore, if a node cannot handle a name, then whp it is able to forward the lookup request to a neighbor that can continue the lookup. One difficulty associated with achieving $\lg n$-fold replication is that the network is dynamic and a node does not know the exact network size. Thus, we need to find a way to enable a node to estimate the network size based on its local state.

For every node u, define the *dimension* of u, denoted by $u.dim$, to be $\max\{i : |\Gamma(u,i)| \ge c \cdot i\}$, where c is some sufficiently large constant. We let $u.similar$ denote $\Gamma(u, u.dim)$ and we call the nodes in $u.similar$ (except u itself) the *similarity neighbors* of u. We modify SPRR as follows in order to achieve fault tolerance:

1. A node u maintains pointers to all the nodes in $u.similar$, as well as the order in which they appear on the locality ring. Thus, we view $u.similar$ as a circular list and define $u.similar.next(v)$ to be the first node in $u.similar$ clockwise from v.

2. A name is replicated at all the nodes in $u.similar$, where u is a node in the best match set of the name.

With these modifications, SPRR provides a significant level of fault tolerance, as is evidenced by Lemmas 4.1 and 4.2 below.

[1] In fact, Chernoff bounds implies that if a name is replicated at $\Omega(\lg n)$ nodes, then $\Omega(\lg n)$ of these nodes are up whp.

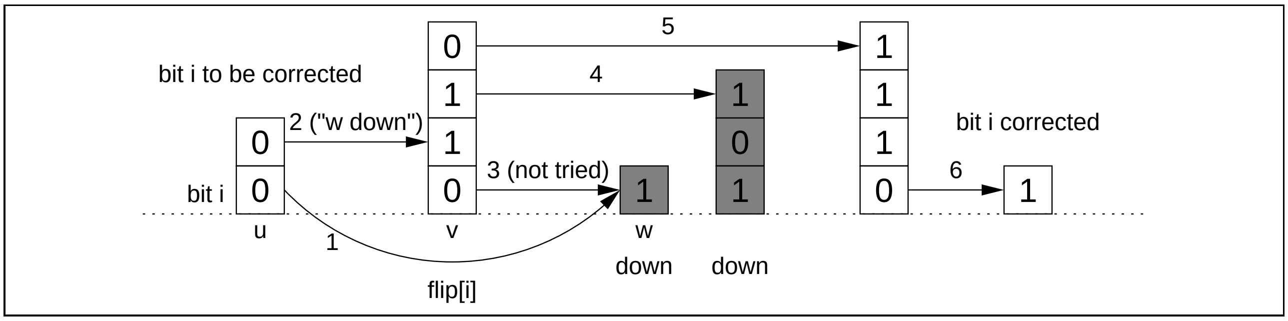

Figure 6: Fault-tolerant lookup example.

4.2 Fault-Tolerant Lookup

Fault-tolerant lookup is a simple extension of fault-free lookup. The main augmentation is handling down nodes. The idea is “bypassing” down neighbors by successively trying higher bit flip neighbors or similarity neighbors. Roughly speaking, when a node u needs to correct bit i but detects that $u.\mathit{flip}[i]$ (also denoted by w for simplicity) is down, it successively tries its higher bit flip neighbors until an up one, $u.\mathit{flip}[j]$ (also denoted by v for simplicity), is found. The lookup request is then forwarded to v, which tries to correct bit i. When forwarding the lookup request to v, node u piggybacks the pointer to w, so that v does not need to reprobe w if $v.\mathit{flip}[i] = w$, in which case v tries to forward the request to one of its upper bit flip neighbors, starting with bit $j + 1$. If a node has exhausted all of its flip neighbors, then it successively tries its similarity neighbors.

Figure 6 shows an example of correcting a single bit in a fault-tolerant lookup. Figure 7 shows the code for fault-tolerant lookup. We will show in Section 4.3 that whp this code successfully completes the lookup operation using $O(\lg n)$ messages. However, this code has the defect that, with (polynomially) small probability, a fault-tolerant lookup may not terminate. For example, suppose that the bit to be corrected is bit i, and the lookup request reaches node u, which has exhausted all of its flip neighbors and is about to try its similarity neighbors. Further assume that u and its similarity neighbors all have the same dimension value, and their bit-i neighbors are all down. Under such a circumstance, the lookup request will be forwarded among u and its similarity neighbors forever. Here we briefly sketch a few possible remedies to this problem:

1. Before forwarding a lookup request to a similarity neighbor, a node can first query that neighbor to make sure that it can successfully correct the bit (i.e., both the similarity neighbor and its appropriate flip neighbor are up). If none of the similarity neighbors can correct the bit, then the lookup fails.

2. One can use a TTL (time-to-live) value to control how many times a lookup request can be forwarded before failure is reported.

3. One can use a TTL value to control how many high messages can be sent before failure is reported.

4.3 Analysis

We first use standard Chernoff bound arguments to establish the following lemmas.

Lemma 4.1 *For every node u, $u.dim = \lfloor \lg n - \lg\lg n - O(1) \rfloor$ whp.*

Lemma 4.2 *For every node u, $|u.similar| = \Theta(\lg n)$ whp.*

```
□ true → {fault-tolerant lookup}
    a := arbitrary name;
    send resolve(me, a, 0) to me;

□ rcv resolve(x, a, i) →
    do (id[i] = a[i]) → i := i + 1;
    od;
    if (a ∈ db ∨ flip[i] = nil) → send reply(a, db.find(a)) to x;
    □ (a ∉ db ∧ flip[i] ≠ nil) →
        if (up(flip[i])) → send resolve(x, a, i + 1) to flip[i];
        □ (¬up(flip[i])) → send low(x, flip[i], a, i, i + 1) to me;
        fi;
    fi;

□ rcv low(x, y, a, i, j) →
    if (flip[i] ≠ y) →
        if (up(flip[i])) → send resolve(x, a, i + 1) to flip[i];
        □ (¬up(flip[i])) → send low(x, flip[i], a, i, j) to me;
        fi;
    □ (flip[i] = y ∧ j < dim) →
        if (up(flip[j])) → send low(x, y, a, i, j + 1) to flip[j];
        □ (¬up(flip[j])) → send low(x, y, a, i, j + 1) to me;
        fi;
    □ (flip[i] = y ∧ j ≥ dim) → send high(x, y, me, a, i, j) to me;
    fi;

□ rcv high(x, y, z, a, i, j) →
    z := similar.next(z);
    if (z = me) → send reply(a, nil) to x;
    □ (z ≠ me ∧ up(z)) → send low(x, y, a, i, j) to z;
    □ (z ≠ me ∧ ¬up(z)) → send high(x, y, z, a, i, j) to me;
    fi;
```

Figure 7: Code for fault-tolerant lookup.

Lemma 4.3 *For all nodes u and v, $|u.dim - v.dim| \leq 1$ whp.*

We next prove some efficiency and locality properties of fault-tolerant lookups in SPRR, which are stated in the following two theorems.

Theorem 7 *Every fault-tolerant lookup takes $O(\lg n)$ messages whp.*

Theorem 8 *The expected total distance traveled by all the messages in a fault-tolerant lookup is $O(n/\lg n)$.*

The proofs of these two theorems are significantly more involved than those presented earlier in the paper. We only sketch our main proof ideas here.

We first introduce a few definitions. A message is said to be *low* if it is from a node to one of its flip neighbors. A low message is said to be *i-low* if it is from a node to its bit i flip neighbor. A message is said to be *high* if it is from a node to one of its similarity neighbors. A high message is 1-high if it is sent from a node u to its next node in $u.similar$, 2-high if it is sent to the next node of the next node in $u.similar$, et cetera. A lookup is divided into *phases*, where phase i consists of the messages associated with correcting bit i.

Our approach to proving Theorem 7 is as follows. At a high level, when a node u wants to correct bit i, it first tries to do so using a path of length one, that is, by forwarding the lookup to $u.\mathit{flip}[i]$. If $u.\mathit{flip}[i]$ is down, our fault-tolerant lookup proceeds by successively trying to correct bit i by using paths of length two, where the first hop on the path leads to a node matching u in bits 0 through i (or higher) and the second hop corrects bit i. We now state a key technical lemma.

Lemma 4.4 *Each successive path considered in a given phase has a constant probability of terminating the phase.*

Proof: We provide a sketch of the proof only. Fix a path P that we are about to explore. We claim that with constant probability, all of the nodes in P are up. To establish this claim, first note that the nodes in path P have never been previously examined, and hence we can view each of them as having a constant probability of being up, independent of the previous history of the lookup. Unfortunately, this argument alone is insufficient to establish the desired lemma. The remaining difficulty is associated with the the case where the first hop of the path brings us to a node u whose bit i neighbor v is already known to be down because we previously attempted to terminate phase i by sending an i-low message to v. In such a case, our algorithm abandons this path P without attempting to send a message along the second hop; instead, we initiate a new two-hop path. It remains to prove that we do not expend a large number of messages, and travels a large distance, due to repeatedly abandoning such two-hop paths at the intermediate nodes.

How can we rule out this scenario? We now argue that there is a constant probability that the bit i neighbor of an intermediate node on a two-hop path P is a node that we have not previously encountered in the lookup, from which it follows that the path P is not abandoned at the intermediate node (since $u.\mathit{flip}[i] \neq w$). The intuition underlying this claim is that the first message on any two-hop path in phase i is either a j-low message for some $j > i$ or a high message. In either case, the expected distance traveled by such a message is greater than that of an i-low message. This observation can be used to show that with constant probability, the first message of the two-hop path passes over any node v that we might have previously determined to be down when sending an i-low message. (In our formal proof of this claim, we defer revealing the precise location of the node v until it is passed over by some message to an up node.) Hence, there is a constant probability that the bit i flip neighbor of the intermediate node of path P is a node that we have not previously encountered. ■

With Lemma 4.4 in hand, it is straightforward to establish Theorem 7 using a standard Chernoff bound argument.

Lemma 4.4 also gives us a good start on establishing Theorem 8. At a high level, the main difference between the proofs of Theorems 7 and 8 is that in the latter case we need to account for the different kinds of messages (i.e., i-low and i-high messages, for various values of i) separately, because the expected distances that they travel vary. Lemma 4.4 can be used to show that a lookup uses expected $O(1)$ i-low messages for any given i, and expected $O(q^i)$ i-high messages for any given i. Theorem 8 follows easily once we establish the following claim: the expected distance traveled by any i-low message is $O(2^i)$ and the expected distance traveled by any i-high message is $O(i \cdot n/\lg n)$. In what follows, we sketch a proof of this claim.

Note that the bit i flip neighbor of a given node u is the first node v clockwise from u such that $match(u, v) = i$. It follows that if each node on the ring has a random ID, then the expected distance from u to its bit i flip neighbor is $O(2^i)$. A similar argument shows that the expected distance traveled by an i-high message is $O(i \cdot n/\lg n)$. Unfortunately, there is a technical obstacle that prevents us from directly applying this simple approach to bound the expected distance of the messages sent during a lookup. The difficulty is that as the lookup algorithm unfolds, information concerning the node IDs is revealed. Consequently, when a particular message is sent by the algorithm, we cannot assume that all of the node IDs are still random. In particular, there are three kinds of information that we learn about the node IDs as the algorithm proceeds. Below we discuss each of these kinds of information in turn and sketch how to bound their effect on our analysis.

1. For any node u that has received a previous message (or would have received a previous message but was determined to be down), we know that the ID of u is inconsistent with any prefix that we will subsequently search for. Thus, if we happen to encounter such a node u while searching for the destination of a subsequent message, the probability that u is the desired destination is 0 (as opposed to, e.g., $\Theta(2^{-i})$ for an i-low message). Since Theorem 7 tells us that whp there are $O(\lg n)$ such nodes u, it is straightforward to argue that the total extra distance incurred by retraversing these nodes is $O(\lg^2 n) = o(n/\lg n)$ whp.

2. For any node u that has been passed over in a search for the destinations of one or more previous messages, we know that the ID of u does not match certain prefixes. Fortunately, this information only tends to (slightly) increase the probability that such a node u is a match for a subsequent search.

3. Finally, a more subtle issue is that as the algorithm unfolds, we learn information concerning the dimensions of certain nodes. This information is global in nature as it tells us something about the total number of nodes matching a node u in a certain prefix. For example, if we learn that the dimension of node u is 10, then we know that $|\Gamma(u, 10)| \geq 10c$ and $|\Gamma(u, 11)| < 11c$, where c is the constant appearing in the definition of dimension (see Section 4.1). But note that Lemma 4.3 tells us that for a given value of n, every node has the same dimension, to within one, whp. This implies that learning some (or all) of the node dimensions is unlikely to bias the probability of occurrence of any given prefix by more than a constant factor.

5 Related Work

Early generations of peer-to-peer networks use unscalable approaches for name resolution. For example, Napster [19] uses a central directory, Gnutella [8] uses flooding, and Freenet [5] uses heuristic search.

Besides PRR, other name resolution schemes include Chord [27], CAN [23], and Viceroy [16]. PRR-like topologies are later also used by Tapestry [29], Pastry [25], and Kademlia [17]. Several systems have been built on top of these schemes: OceanStore [13] and Bayeux [30] on Tapestry, PAST [7] and SCRIBE [26] on Pastry, and CFS [6] on Chord.

Besides hypercubes, shuffle-exchange networks [16] or de Bruijn graphs [22] can also be used for name resolution. For example, Viceroy [16] uses shuffle-exchange networks, in which every node maintains only a constant, instead of logarithmic, number of neighbors. The advantage of constant degree is reduced cost for joins and leaves. However, the disadvantages are: (1) locality is exploited less effectively because there are fewer choices for a neighbor, (2) the network is vulnerable to being partitioned because each node only has a constant degree, and (3) the constants in the expected running times are higher. However, an important research issue is ensuring the correctness of concurrent name resolution operations. In this respect, constant-degree networks may be easier to reason about. Thus, the pros and cons of such constant-degree constructions merit further investigation.

Chord works by arranging nodes and names on an ID ring. A name is stored at a node immediately succeeding the name on the ID ring. Apart from a predecessor and successor pointer, each node maintains a logarithmic number of finger pointers. A finger pointer points to the first node succeeding a certain point on the ID ring. The finger pointers enable efficient name resolution, while the (possibly multiple) predecessor and successor pointers ensure fault tolerance.

CAN works by mapping nodes and names to a d-dimensional unit space. Each node is assigned a region in the space and is responsible for resolving the names mapped to that region. For a network with n nodes, a lookup takes $O(d \cdot n^{1/d})$ hops. Thus, to achieve logarithmic scalability, CAN needs to set $d = \lg n$, which may not be feasible without a good anticipation of n or if n changes dramatically during the lifetime of the network.

The importance of locality is now widely recognized and most name resolution schemes go to significant lengths to exploit locality, be it rigorously [11, 12, 20] or heuristically [4, 24, 28]. As discussed in Section 1, there is a tradeoff between simplicity and effectiveness of exploiting locality, and SPRR attempts to exploit locality without sacrificing simplicity.

At a high level, SPRR, especially with the implementation maintaining only predecessors and successors, bears some resemblance to a skip list [21], a randomized dictionary data structure whose applications to peer-to-peer computing has recently gained attention. For example, Karger and Ruhl [12] have proposed a data structure called *metric skip list* to solve the nearest neighbor problem on growth-restricted metric spaces. In the process of finalizing this technical report, we have learned of two other independent research efforts involving skip-list-like structures: skip graphs [1] and hyperrings [2]. While similar to SPRR at the high level, the primary design objectives underlying the work in [1] and [2] (e.g., range queries, fault-tolerant connectivity, and repairability) are different from those of the present paper, and consequently the details of the constructions and analyses differ substantially. Going forward, it would be interesting to exhibit a single skip-list-like topology that enjoys most or all of the various strong theoretical properties established here and in [1, 2].

6 Future Work

Compared to PRR, a drawback of SPRR is that the average edge length is increased, due to the additional constraint on neighbor selection. Bounding this increase is an open problem. Ideally, we would like to bound it by constructing a ring such that the total edge length of SPRR is only a constant factor larger than that of PRR.

As discussed in Section 1, simplicity not only is a desirable feature of system design, but also helps reasoning about the correctness of a name resolution scheme. Maintaining the neighbor tables is a complicated task. When many joins and leaves happen concurrently, it is not clear whether the neighbor tables will remain in a "good" state. This problem, however, has not been adequately addressed by current research. The problem is much easier if the network is allowed to have some "locking" mechanism. However, for performance reasons, it is desirable that name resolution operations be *non-blocking* [10], that is, slow operations cannot prevent other operations from making progress. We plan to implement a non-blocking name resolution scheme and to prove the correctness of the implementation. The work of Blumofe *et al.* [3] suggests that proving the correctness of such non-blocking concurrent data structures can be a significant technical challenge. A simple framework like SPRR is an important starting point for our future work.

7 Concluding Remarks

In this paper, we have defined a class of network topologies called hyperdelta networks, and observed that the PRR topology is a random hyperdelta network. To overcome some weaknesses of PRR on arbitrary metric spaces, we have proposed SPRR, a simple variant of PRR. SPRR is considerably simpler than PRR and retains most scalability properties of PRR. When specialized to the case of uniform metric space, SPRR is comparable to Chord in terms of simplicity, and has matching or improved time bounds on name resolution operations. In more general metric spaces, SPRR exploits locality effectively. The ease of exploiting locality comes from the ability to choose neighbors from a set of candidates. In this paper, we have proved the locality property of SPRR on the ring metric. Fault tolerance can be achieved in SPRR without adding much complexity. SPRR employs a novel name replication strategy that ensures lookups remain efficient in a random fault model where each node has a constant probability of being down.

Acknowledgments

The authors would like to thank Frans Kaashoek for bringing to our attention the work of Awerbuch and Scheideler [2], and thank Christian Scheideler for bringing to our attention the work of Aspnes and Shah [1].

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2003. To appear.

[2] B. Awerbuch and C. Scheideler. Self-repairing and searchable distributed data structures, November 2002. Under submission.

[3] R. D. Blumofe, C. G. Plaxton, and S. Ray. Verification of a concurrent deque implementation. Technical Report TR–99–11, Department of Computer Science, University of Texas at Austin, June 1999.

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, October 2001.

[7] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[8] Gnutella. Available at http://gnutella.wego.com.

[9] M. G. Gouda. *Elements of Network Protocol Design*. John Wiley & Sons, 1998.

[10] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.

[11] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed data location in a dynamic network. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, August 2002.

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

[13] 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. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, November 2000.

[14] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[15] X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proceedings of the 2nd Workshop on Principles of Mobile Computing*, pages 82–89, October 2002.

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

[17] P. Maymounkov and D. Maziéres. Kademlia: A peer-to-peer information system. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[18] D. S. Mitrinović. *Analytic Inequalities*. Springer-Verlag, Berlin, 1970.

[19] Napster. Available at http://www.napster.com.

[20] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[21] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[22] R. Rajaraman, A. W. Richa, B. Vöcking, and G. Vuppuluri. A data tracking scheme for general networks. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 247–254, July 2001.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, 2001.

[24] S. Ratnasamy, M. Hanley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, June 2002.

[25] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[26] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event-notification infrastructure. In *Proceedings of the 3rd International Workshop on Network Group Communications*, pages 30–43, November 2001.

[27] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, 2001.

[28] B. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[29] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, April 2001.

[30] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and OS Support for Digital Audio and Video (NOSSDAV)*, pages 11–20, July 2001.