On Name Resolution in Peer-to-Peer Networks*

Xiaozhou Li C. Greg Plaxton

ABSTRACT

An efficient name resolution scheme is the cornerstone of any peer-to-peer network. The name resolution scheme proposed by Plaxton, Rajaraman, and Richa, which we hereafter refer to as the PRR scheme, is a scalable name resolution scheme that also provides provable locality properties. However, since PRR goes to extra lengths to provide these locality properties, it is somewhat complicated. In this paper, we propose a scalable, locality-aware, and fault-tolerant name resolution scheme which can be considered a simplified version of PRR. Although this new scheme does not provide as strong locality guarantees as PRR, it exploits locality heuristically yet effectively.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—network topology; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—routing and layout

General Terms

Algorithms, Performance, Reliability

Keywords

peer-to-peer networks, name resolution

1. INTRODUCTION

(Note: This version of the paper includes corrections to a few minor errors that appeared in the original workshop version. See Appendix A for a complete list of the differences between this version and the workshop version.)

An efficient name resolution (or distributed data lookup) scheme is the cornerstone of any peer-to-peer network. Given the name of a data item, the task of name resolution is to find the location of the item. (Although this is what name resolution is normally used for, a more general definition is: given a name, determine the value to which the name maps.) Since a peer-to-peer network can potentially have a large number of nodes, scalability is a fundamental requirement for name resolution. Although the usual measurement for scalability is the number of (application-level) hops between any two nodes, the distance traveled by each hop, namely, locality, 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).

Considerable research effort has been invested in the design of scalable name resolution schemes and several such schemes have been proposed, including the PRR scheme proposed by Plaxton *et al.* [13], Tapestry [21], Pastry [17], Chord [19], and CAN [15].

Despite much work, certain aspects of name resolution merit further investigation. Consider the PRR scheme. PRR uses a variant of hypercubic routing as the basic routing method and this method is later also used by Tapestry and Pastry. Besides guaranteeing a logarithmic number of hops between any two nodes, PRR provides provable locality properties for a restricted class of metric spaces. Recent research results reduce the restriction on the metric spaces [8, 9]. Providing these properties on a general metric space, however, remains an open problem.

In order to exploit locality, PRR maintains a logarithmic-sized collection of neighbor pointers at each node u. Each neighbor of u is the nearest node to u with an ID matching that of u in a certain number of bit positions. Maintaining these neighbor pointers is a nontrivial task, especially if the distance function is changing, or if nodes are frequently joining or leaving the network. On the other hand, Chord [19], which disregards locality, is relatively simple. Thus, there is an apparent tradeoff between the simplicity of a name resolution scheme and its ability to exploit locality. Since extreme approaches like PRR or Chord both have undesirable features, is it possible to design a scheme that is simple yet still effectively exploits locality?

In this paper, we present results that improve on various aspects of name resolution. Built on top of previous work, these results will prove useful for better understanding of the various issues of name resolution.

We first consider PRR without locality (i.e., we specialize PRR to the case of a uniform metric space). The resulting scheme, which we call SPRR (simplified PRR), is simple and scalable. Compared to Chord, SPRR is not only arguably simpler, but also 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

*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.

POMC'02, October 30-31, 2002, Toulouse, France.

probability (whp), whereas, given the Chord definition of fingers, the join operation in Chord requires $\Omega(\lg^2 n)$ time to succeed 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.) 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 mobile environment, the ability to quickly add or remove a node from the network is particularly important.

We then show that locality can be exploited in SPRR without adding much complexity. We argue that SPRR is likely to have good locality properties on any metric space, and we give rigorous locality properties of SPRR 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 in the 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 SPACES

In this section, we 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. However, for real peer-to-peer applications, locality and fault tolerance have to be added to SPRR. In Sections 3 and 4, we will show how to exploit locality and achieve fault tolerance without adding much complexity.

The rest of this section is organized as follows. Section 2.1 presents the basic organization of SPRR. Section 2.2 presents the basic name resolution operations of SPRR. Section 2.3 analyzes the properties of SPRR.

2.1 Basic Organization

Every node and every object has an identifier (ID), which is a fixed-length random binary string. Node IDs and object IDs are drawn from the same space, which can be thought of as a ring, called the *ID ring*, by arranging the IDs according to their lexicographic order (and wrapping around). Object IDs are also called *names* in this paper. IDs can be generated using a hash function. For the purpose of our analysis, we view IDs as a sequence of random bits. In practice, the length of an ID, denoted by ℓ, is chosen to be long enough (say, 128 bits) so that the chance of having two identical IDs is negligible. ID bits are numbered from left to right as bit 0 to to bit $\ell - 1$.

We first introduce a few notations. Let n be the number of nodes in the network; ℓ be the length of an ID; $u[i]$ be bit i of u, where $0 \le i < \ell$; $match(x, y)$ be the length of the longest common prefix shared by the IDs of x and y; $F(u, i)$ be the set $\{v \mid match(u, v) = i\}$; $N(\beta)$ be the set of nodes prefixed by bit string β; $\Gamma(u, i)$ be the set $\{v \mid match(u, v) \ge i\}$. At times, we identify a node with its ID when no confusion could arise.

Every node u maintains three types of neighbors:

- *Forward neighbors.* The bit i *forward neighbor*, denoted as $u.f[i]$, where $0 \le i < \ell$, is a node in $F(u, i)$. If $|F(u, i)| = 0$, then $u.f[i] = \text{NULL}$. Our convention of setting a forward neighbor to NULL in the case no candidate node exists is different from that employed by other schemes. We will discuss the pros and cons of this decision in Section 5. If $|F(u, i)| = 1$, then $u.f[i]$ is the unique node in $F(u, i)$. If $|F(u, i)| > 1$, then a node in $F(u, i)$ is chosen to be $u.f[i]$. Choosing an appropriate node in $F(u, i)$ is an important issue that will be elaborated on below. The total number of forward neighbors of a node is called the *out-degree* of the node.

- *Backward neighbors.* If v is the bit i forward neighbor of u, then u is a bit i *backward neighbor* of v. Note that there is at most one bit i forward neighbor of a node, but there can be multiple bit i backward neighbors. We let $u.B[i]$ denote the set of bit i backward neighbors of u. The total number of backward neighbors of a node is called the *in-degree* of the node.

- *Predecessor and successor.* Borrowing an idea from Chord, we introduce two more neighbors for each node u: (1) a *predecessor*, which is the node immediately preceding u on the ID ring, and (2) a *successor*, which is the node immediately succeeding u on the ID ring. For fault-free name resolution, each node maintains exactly one predecessor and one successor. For fault-tolerant name resolution (see Section 4), each node maintains multiple predecessors and successors. The total number of neighbors of a node is called the *degree* of the node.

As discussed above, when $|F(u, i)| > 1$, we need to choose a "good" node in $F(u, i)$ to be the bit i forward neighbor. The freedom to choose a node from a set of candidates is an important feature and advantage of PRR. Different criteria, based on locality, fault tolerance, or other considerations, can be used to make the choice. Perhaps the simplest criterion is to choose an arbitrary node in $F(u, i)$. This criterion, however, can result in a high in-degree for a node (e.g., all of the nodes prefixed by 1 can have the same bit 0 forward neighbor). Nodes with high in-degree or out-degree are undesirable because when such a node joins or leaves the network, many neighbor pointers have to be modified. On the other hand, maintaining sufficient degrees for a node is necessary for fault tolerance, since a node with low degree is vulnerable to being isolated from the rest of the network. For example, consider a random fault model where each node has a constant probability of being down. In such a model, in order to guarantee that some of the neighbors of each node are up whp, each node needs to maintain a logarithmic number of neighbors.

We propose the following approach to choosing a forward neighbor from a set of candidates. We first place the nodes on a *logical ring*, which is independent of the ID ring, and define $u.f[i]$ to be the closest node clockwise to u. This definition of forward neighbors avoid the problem of having a large in-degree for a node. We show in Section 2.3 that this approach results in both logarithmic in-degree and out-degree whp. We also show in Section 3.3 that locality can be exploited if the distance in the logical ring is correlated with the distance in the metric space. Finally, node u maintains the nodes in $u.B[i]$ in the clockwise order that they appear on the logical ring.

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 $u.name\_db$, to store the names for which it is responsible. SPRR assigns handlers as follows. Let the *best match set* of a name α, denoted by $M(\alpha)$, be the set $\{u \mid \forall v : match(u, \alpha) \geq match(v, \alpha)\}$. We call $match(u, \alpha)$, where $u \in M(\alpha)$, the *depth* of the best match set. When a name α is inserted, the insert request is forwarded until a node in $M(\alpha)$ is reached. This node is designated as the handler of the name. When a name is looked up or deleted, additional work has to be done to locate the handler of the name. We describe the details of finding the handler in Section 2.2.

2.2 Basic Operations

With the above definitions, we are now ready to present the basic operations of SPRR. SPRR supports the following two types of operations: (1) *name operations*, including lookup (looking up a name), insert (inserting a name), and delete (removing a name); and (2) *network operations*, including join (adding a node to the network), and leave (removing a node from the network).

Central to the name operations is finding the handler of a name. SPRR divides the process into two phases: (1) the *bit-correcting* phase, which is used to reach a node in the best match set; (2) the *walking* phase, which is used to traverse some of the nodes in the best match set by following predecessor and successor pointers. For an insert operation, only the bit-correcting phase is needed, while for a lookup or delete operation, both phases are needed.

- *Insert.* When a name is inserted, we only need to find an arbitrary node in the best match set. The process of finding such a node is by bit-correcting: a node forwards the insert request to a forward neighbor that matches the name in more bits than the node itself, until no such neighbors can be found. When the insert request cannot be forwarded further, a node in the best match set has been reached, and that node is the handler of the name. The code for finding a node in the best match set is shown in Figure 1, and the code for the insert operation is shown in Figure 2.

- *Lookup.* To look up a name, a node first needs to find the handler of the name. The process of finding the handler includes both the bit-correcting phase and the walking phase. The walking phase is needed because the first node reached in the best match set during the lookup operation may not be the handler of the name. Thus, we need to traverse the best match set until we locate the handler. The code for finding the handler is shown in Figure 1, and the code for the lookup operation is shown in Figure 2.

- *Delete.* Deleting a name is largely similar to looking up a name: first the handler is found, then the name is removed from the local name database of the handler. The code for the delete operation is shown in Figure 2.

- *Join.* When a new node joins, the new node first contacts an arbitrary node, called the *contact*. We assume that some external mechanism enables the new node to find a contact, an assumption made by many other schemes. (As will be discussed in Section 3.3, 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 as the contact.) The new node then generates its own ID and places itself immediately before the contact on the logical ring. The new node then builds its own neighbor table by consulting a sequence of nodes, starting from the contact. During this process, the existing neighbor tables of current nodes are also updated. The code for the join operation is shown in Figure 3.

- *Leave.* When a node leaves, it needs to inform its backward neighbors to change their forward neighbor pointers. The code for the leave operation is shown in Figure 4.

```
u.find_bms(α)
    k ← match(u, α);
    if (u.f[k] ≠ NULL) then
        return u.f[k].find_bms(α);
    else return u;

u.find_handler(α)
    v ← u.find_bms(α);
    return v.walk(α);

u.walk(α)
    w ← u;
    while (match(w, α) = match(u, α)) do
        if (α ∈ w.name_db) then
            return w;
        else w ← w.pred;
    w ← u.succ;
    while (match(w, α) = match(u, α)) do
        if (α ∈ w.name_db) then
            return w;
        else w ← w.succ;
```

Figure 1: Code for finding the best match set and handler for name α.

```
u.insert(α, value)
    v ← u.find_bms(α);
    v.name_db.add(α, value);

u.lookup(α)
    v ← u.find_handler(α);
    return v.name_db.find(α);

u.delete(α)
    v ← u.find_handler(α);
    v.name_db.remove(α);
```

Figure 2: Code for lookup, insert, and delete.

2.3 Analysis

In this section, we analyze the properties of SPRR. Our main result is that all operations in SPRR, including lookup, insert, delete, join, and leave, take $O(\lg n)$ constant-size messages (or application-level hops) whp. For the two network operations join and leave, this represents a significant improvement over the $O(\lg^2 n)$ message

```
// u is the new node, v is the contact.
u.join(v)
    for (i ← 0 to ℓ − 1) do
        if (u[i] = v[i]) then
            u.f[i] ← v.f[i];
            u.B[i] ← v.B[i];
            v.B[i] ← ∅;
            foreach (w ∈ u.B[i]) do
                w.f[i] ← u;
        else
            u.f[i] ← v;
            v.B[i].append(u);
            if (v.f[i] ≠ NULL) then
                w ← v.f[i];
                for (j ← 0 to w.B[i].size − 1) do
                    x ← (w.B[i])[j];
                    if (x ≠ v) then
                        u.B[i].append(x);
                        v.B[i].remove(x);
                    else break ;
                foreach (w ∈ u.B[i]) do w.f[i] ← u;
            else
                w ← v;
                while (match(w, u) = i) do
                    w.f[i] ← u;
                    w ← w.pred;
                if (u[i] < v[i]) then
                    u.slice_between(w, w.succ);
                w ← v.succ;
                while (match(w, u) = i) do
                    w.f[i] ← u;
                    w ← w.succ;
                if (u[i] > v[i]) then
                    u.slice_between(w.pred, w);
                break ;

u.slice_between(v, w)
    u.pred ← v; u.succ ← w;
    v.succ ← u; w.pred ← u;
```

Figure 3: Code for join.

```
u.leave()
    for (i ← 0 to ℓ − 1) do
        if (u.f[i] ≠ NULL) then
            u.f[i].B[i].remove(u);
            if (u.f[i].B[i].size = 0) then
                v ← u.f[i].f[i];
            else v ← (u.f[i].B[i])[0];
            if (v = u) then
                v ← NULL;
            foreach (w ∈ u.B[i]) do
                w.f[i] ← v;
            if (v ≠ NULL) then
                v.B[i].prepend(u.B[i]);
```

Figure 4: Code for leave.

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 $1/2^{i+1}$ 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 α, 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

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

The sum of the first $\lg n$ terms is bounded by $\lg n$. The sum of the other terms is $O(1)$ because the ratio of the successive term to the preceding term is strictly less than one. Thus $\mathrm{E}[X] = \lg n + O(1)$. □

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

PROOF. Without loss of generality, assume the name α to be looked up is all 0's. Let n_j be the number of nodes prefixed by j 0's. Consider the maximum k such that $n_k \geq i$. For $|M(\alpha)|$ to be equal to i, it is necessary that $n_k = i$ and $n_{k+1} = 0$. Note that $\Pr[n_{k+1} = 0 \mid n_k = i] = 1/(2^i - 1)$. Thus $\Pr[|M(\alpha)| = i] = O(2^{-i})$, and hence $\mathrm{E}[|M(\alpha)|] = O(1)$.

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 expection 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* $\lg n + O(\sqrt{\lg n})$ *forward neighbors whp.*

PROOF. Starting from bit 0, we divide the ID of the node under consideration into three segments A, B, and C, each with length $\lg n$, $c \lg n$, and $\ell - (c+1)\lg n$. Let X_A, X_B, and X_C be the number of forward neighbors in these segments. Then $X_A \leq \lg n$, $X_B = O(\sqrt{\lg n})$ whp, and $X_C = 0$ whp. Thus, the number of forward neighbors is at most $\lg n + O(\sqrt{\lg n})$ whp. □

LEMMA 2.6. *For all sufficiently large positive constants c, for all* $i \geq c \lg n$, *and for all nodes* u, $u.f[i] = \mathrm{NULL}$ *whp.*

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

LEMMA 2.7. *Every node has* $O(\lg n)$ *backward neighbors 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 the subsequent nodes prefixed by 0, once we see a 0 on bit 1, we start inspecting bit 2 of the subsequent nodes prefixed by 00, and so forth. We keep inspecting until we return to the node u. The key observation is that the set of nodes inspected in this process is the set of backward neighbors of u. Furthermore, by Lemma 2.6, no node has a 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 $O(\lg n)$. □

THEOREM 3. *Every node has* $O(\lg n)$ *neighbors whp.*

PROOF. Immediate from Lemmas 2.5 and 2.7. □

THEOREM 4. *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 Theorem 3. □

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 its definition of forward neighbors. Unlike Chord, which defines a finger to point to the first node following a certain point on the ID ring, PRR (and SPRR) defines a forward 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.

Section 3.1 first proposes our name replication strategy; Section 3.2 explains heuristically 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 α, when the insert request reaches a node u in the best match set, u replicates α at r nodes around itself on the ID ring that match α better than the rest of the nodes. Note that the replication can be achieved by simply following the predecessor and successor pointers. We call these r nodes the *replication set* of α, denoted by $R(\alpha)$. In other words, $R(\alpha)$ is a set such that $|R(\alpha)| = r$ and for any $v \in R(\alpha)$ and $w \notin R(\alpha)$, $match(v,\alpha) \geq match(w,\alpha)$.

This replication strategy requires a node check its local name database before forwarding the name to a neighbor, because now a name can be stored at multiple nodes.

3.2 Heuristic Exploitation of Locality

SPRR's ability to exploit locality originates from PRR's flexibility to choose a good forward neighbor from a set of candidate nodes. For example, consider the process of choosing the bit 0 forward 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 properties of SPRR on the ring metric, where the distance between two nodes is the distance between them on the 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 2.1, a node chooses a forward 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 5. *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 d_1 and d_2 denote the distances traveled in the bit-correcting and walking phases, respectively. Each successive hop in the bit-correcting phase tends to cover geometrically greater distance so that the total distance is dominated by that of the last hop. The last hop covers expected $O(n/r)$ distance because, with r-fold replication, the copies tend to be spaced $\Theta(n/r)$ apart on the locality ring. Thus, $\mathrm{E}[d_1] = O(n/r)$. Moreover, when $|M(\alpha)| \leq r$, $d_2 = 0$; when $|M(\alpha)| > r$, $d_2 = O(n(|M(\alpha)| - r))$. By

Lemma 2.4, we have $\Pr[|M(\alpha)| = i] = O\left(1/2^i\right)$. Therefore, $\mathrm{E}[d_2] = O(n/r)$. □

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 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 SPRR

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. (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.) 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 $\max\{i \mid \Gamma(u,i) \geq c \cdot i\}$, 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. Instead of maintaining only one predecessor and one successor, 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 $next(u.similar, v)$ to be the first node in $u.similar$ as we proceed 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.

4.2 Fault-Tolerant Lookup

Fault-tolerant lookup is a simple extension of fault-free lookup. The main augmentation is handling down nodes. The code for fault-tolerant lookup is shown in Figure 5. In order to look up a name α, u invokes $u.\mathit{ft\_lookup}(\alpha, 0, \mathrm{NULL})$. The main idea is "bypassing" down neighbors by trying higher bit neighbors or similarity neighbors. The key property is that a node is never probed more than once. This property will prove crucial for our analysis.

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.*

```
u.ft_lookup(α, j, w)
    if (α ∈ u.name_db) then
        return u.name_db.find(α);
    i ← match(u, α);
    if (u.f[i] ≠ w ∧ up(u.f[i])) then
        return u.f[i].ft_lookup(α, i + 1, NULL);
    w ← u.f[i];
    j ← max(j, i + 1);
    while (j < u.dim) do
        if (up(u.f[j])) then
            return u.f[j].ft_lookup(α, j + 1, w);
        j ← j + 1;
    v ← next(u.similar, u);
    while (v ≠ u) do
        if (up(v)) then
            return v.ft_lookup(α, u.dim, w);
        v ← next(u.similar, v);
    return FAIL;
```

Figure 5: Code for fault-tolerant lookup.

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

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 SPRR, which are stated in the following two theorems.

THEOREM 6. *Every fault-tolerant lookup takes $O(\lg n)$ messages whp.*

THEOREM 7. *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. Due to space constraints, we only sketch our main proof ideas.

We first introduce a few definitions. A message is said to be *low* if it is from a node to one of its forward neighbors. A low message is said to be *i-low* if it is from a node to its bit i forward 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 6 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 $u.f[i]$. If $u.f[i]$ 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 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 our algorithm has the property that any node is the recipient of at most one message throughout the course of the lookup. It follows 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 conjunct $u.f[i] \neq w$ appearing on the fifth line of the fault tolerant lookup code. This conjunct deals with 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 the conjunct $u.f[i] \neq w$ evaluates to true). 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 neighbor of the intermediate node of path P is a node that we have not previously encountered. $\square$

With Lemma 4.4 in hand, it is straightforward to establish Theorem 6 using a standard Chernoff bound argument.

Lemma 4.4 also gives us a good start on establishing Theorem 7. At a high level, the main difference between the proofs of Theorems 6 and 7 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 $O\left(q^i\right)$ i-high messages for any given i. Theorem 7 follows easily once we establish the following claim: the expected distance traveled by any i-low message is $O\left(2^i\right)$ and the expected distance traveled by any i-high message is $O\left(i \cdot n/\lg n\right)$. In what follows, we sketch a proof of this claim.

Note that the bit i neighbor of a given node u is the first node v encountered in a clockwise search from u such that $match(u, v) = 2^{-i-1}$. It follows that if each node on the ring has a random ID, then the expected distance from u to its bit i neighbor is $O\left(2^i\right)$. A similar argument shows that the expected distance traveled by an i-high message is $O\left(i \cdot n/\lg n\right)$. 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 6 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 $O\left(\lg^2 n\right) = o(n/\lg n)$ whp.

2. For any node u that has been passed over in a searching 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 [12] uses a central directory, Gnutella [6] uses flooding, and Freenet [3] uses heuristic search.

Besides PRR, other scalable name resolution schemes include Tapestry [21], Pastry [17], Chord [19], and CAN [15]. Several systems have been built on top of these schemes: OceanStore [10] and Bayeux [22] on Tapestry, PAST [5] and SCRIBE [18] on Pastry, and CFS [4] on Chord. Tapestry and Pastry use similar routing method (i.e., hypercubic routing) as PRR, and add the ability to accommodate node joins and leaves.

Besides hypercubes, shuffle-exchange networks [11] or de Bruijn graphs [14] can also be used for name resolution. For example, Viceroy [11] 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 the 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 easy without a good anticipation of n.

The importance of locality is now widely recognized and most name resolution schemes go to significant lengths to exploit locality, be it rigorously [8, 9, 13] or heuristically [2, 16, 20]. 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.

6. FUTURE WORK

As discussed in Section 1, simplicity is not only a desirable feature on any system design, it 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* [7], 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.* [1] 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 proposed SPRR, a variant of the PRR name resolution scheme. Built on previous work, SPRR has a number of desirable features. 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 without adding much complexity. The ease of exploiting locality comes from the ability to choos neighbors from a set of candidates. In this paper, we have proved the locality propertis 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.

8. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive comments.

9. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

APPENDIX

A. ERRATA

This appendix describes the differences between the current version and the workshop version of the paper.

1. The following sentence has been added to the end of the 8th paragraph of Section 2.1. "Finally, node u maintains the nodes in $u.B[i]$ in the clockwise order that they appear on the logical ring."
2. In Figure 3, the code "$v.B[i].append(u)$;" has been inserted between lines 10 and 11; lines 12–14 have been replaced by the following code:

```
w ← v.f[i];
for (j ← 0 to w.B[i].size − 1) do
    x ← (w.B[i])[j];
    if (x ≠ v) then
        u.B[i].append(x);
        v.B[i].remove(x);
    else break ;
```

3. In Figure 4, the "add" on the last line has been replaced by "$prepend$"; line 5 has been replaced by the following code:

```
if (u.f[i].B[i].size = 0) then
    v ← u.f[i].f[i];
else v ← (u.f[i].B[i])[0];
```

4. There is a typo on the fifth line of the first papagraph of Section 2.3: $O(\lg n)$ should be $O(\lg^2 n)$.
5. In Figure 5, the 0 on line 5 has been replaced by $i + 1$; the $\leq$ on line 7 has been replaced by $<$; the j on line 9 has been replaced by $j+1$; the following code has been inserted bewteen lines 6 and 7: "$w \leftarrow u.f[i];\ j \leftarrow \max(j, i+1);$".