In *Proceedings of the 17th International Symposium on Distributed Computing.*

# Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-Peer Networks

Kirsten Hildrum and John Kubiatowicz

University of California, Berkeley
{hildrum,kubitron}@cs.berkeley.edu

**Abstract.** In this paper, we show that two peer-to-peer systems, Pastry [13] and Tapestry [17] can be made tolerant to certain classes of failures and a limited class of attacks. These systems are said to operate properly if they can find the closest node matching a requested ID. The system must also be able to dynamically construct the necessary routing information when new nodes enter or the network changes. We show that with an additional factor of $O(\log n)$ storage overhead and $O(\log^2 n)$ communication overhead, they can continue to achieve both of these goals in the presence of a constant fraction nodes that do not obey the protocol. Our techniques are similar in spirit to those of Saia *et al.* [14] and Naor and Wieder [10]. Some simple simulations show that these techniques are useful even with constant overhead.

## 1 Introduction

In peer-to-peer systems, all nodes are roughly equal. This equality brings with it the potential for great power: such systems lack a central point of failure and thus could, in principle, be less vulnerable to faults and directed attacks. Unfortunately, achieving this advantage is difficult because peer-to-peer algorithms propagate information widely—greatly expanding the damage wrought by faulty or malicious nodes. In this paper, we take a step forward by showing how two peer-to-peer systems, Pastry [13] and Tapestry [17], can be made tolerant to a limited class of failures and attacks.

Peer-to-peer networks such as Tapestry and Pastry are comprised of many *overlay nodes*, each with a unique random identifier.[1] One of the most important tasks in a peer-to-peer network is *routing*, the ability to pass a message from a source node to a destination node whose ID most closely matches a requested destination. It is this task that we wish to accomplish in the presence of failures.

To accomplish this task, each overlay node maintains a table of connections to a few (normally $O(\log n)$ or sometimes $O(1)$) of the other peer-to-peer nodes, called *neighbors*. These connections are chosen such that routing decisions require only information about the destination ID while keeping the number of hops in overlay path short. In Tapestry and Pastry (and some other systems), an

[1] The name space is chosen large enough that the probability that two randomly assigned names are the same is negligible.

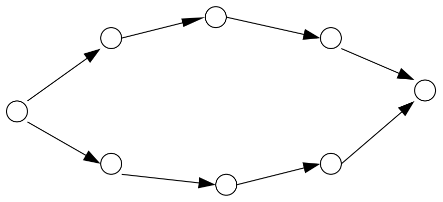

**Fig. 1.** Multi-path diversity: If any node on the path fails, the whole path fails.

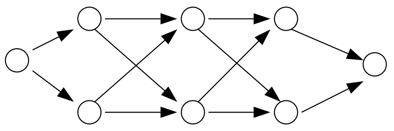

**Fig. 2.** Wide-path diversity: Two nodes must fail at same level to break path.

additional consideration is keeping the network distance short by choosing short links over long ones when possible.

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

The key idea of this paper is to exploit redundancy to tolerate faults, both in building the neighbor table and in routing. There are two basic approaches to routing illustrated in Figures 1 and 2. The first idea (in Figure 1) is to use multiple paths. As long as one path is failure-free, the message will make it from the source to the destination. However, notice that if one node fails in a path, the whole path is useless.

Figure 2 shows a different technique. Instead of two separate paths, this diagram show a single path that is two nodes wide. This means that all the nodes in a given step send to all the nodes in the next step. If any node in one step gets the message, then all the nodes in the next step will also get the message. For the routing to be blocked, at some step, both the top and the bottom nodes must *simultaneously* fail. This provides much greater fault-tolerance per redundant overlay node than multiple paths (even normalizing for bandwidth consumed). We will exploit this technique later.

There is another, orthogonal, routing design decision. The routing outlined above is *recursive*. In recursive routing, the intermediate nodes forward the query on to the next intermediate node. In contrast, some routing algorithms (including the ones described in this paper) are *iterative*. In iterative routing, at each step, the initiating node contacts some other node (or a set of nodes) to get the next hop. The difference is illustrated in Figure 3. Iterative routing is less efficient, but gives the originating node more control, which can be important in a faulty network.

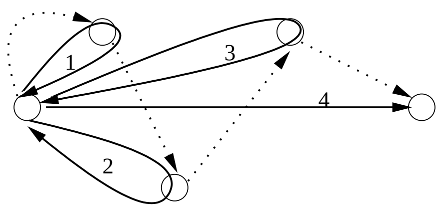

**Fig. 3.** Iterative vs Recursive routing: The source is on the left, the destination is on the right. The solid arrows, labeled by number, represent the iterative path, and the dotted arrows represent the corresponding recursive path.

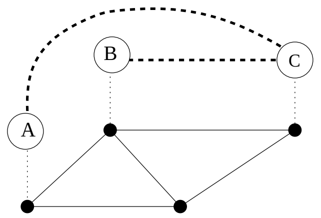

**Fig. 4.** Overlay nodes (hollow circles) with neighbor links (dashed lines) above physical nodes (solid circles) and network links (solid lines). $B$ cannot interfere with messages from $A$ to $C$, but $C$ can interfere with messages from $A$ to $B$.

## 1.1 Related Work

The research community has described a diverse set of peer-to-peer systems [5, 8, 9, 12–14, 16, 17]. Some of the techniques of this paper may apply to other systems.

In a recent paper, Sit and Morris [15] identify some basic categories of attacks on peer-to-peer networks and some general responses to them. Their suggested responses are general enough to apply to most peer-to-peer networks, so they are in some cases not completely specified. Thus, they do not formally prove that their approaches succeed. Two of their concerns are the two problems addressed in this paper—attacks on dynamically building of the routing tables and routing itself. They also suggest that iterative routing may be better in faulty networks.

Douceur in [3] describes the Sybil attack, in which a faulty node generates many IDs and then pretends to be many nodes. He shows that it is practically impossible to prevent this attack without a centralized authority which either explicitly or implicitly distributes IDs. (An IP address is an example of an implicit identifier.)

Castro *et al.* [2] address many the same issues as the Sit and Morris, but as they relate to Pastry in particular. For example, to route securely, Castro *et al.* first route normally, and then perform a routing failure test to determine whether the routing has gone wrong. If their test detects that routing may not have been correct, they retry routing with a secure routing protocol.

Their secure routing protocol uses a different data structure that, unlike the normal routing table, does not take into account network distance. With the assumption that nodes cannot choose their location in the network, and an algorithm to build the normal table correctly with high probability, [2] this backup table may be less important.

Their secure routing technique is essentially that of Figure 1. That is, they send a message from $r$ different starting points. But this technique requires $r$ to

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

be polynomial in $n$ to get a failure probability (over all paths) to be less than a constant. This paper introduces a different technique, using the "wide" paths of Figure 2, that can gives a failure probability of $1/n^c$ when $r$ is only $O(\log n)$ (or $O(\log^2 n)$ in an stronger fault model).

Saia *et al.* [14] and Naor and Wieder[10] also made use of the wide paths depicted in Figure 2. In both cases, they group together nodes into groups of size $O(\log n)$ and use those groups as a single virtual node in the overlay. The network routes correctly as long as one node in each unit follows the protocol. Their routing algorithms are recursive, rather than iterative.

Fiat and Saia [4] construct a butterfly network of virtual nodes where each virtual node is made up of $O(\log n)$ real nodes. (In [14] they make this construction dynamic.) By having more than one starting point, even when the failures are picked to be the worst possible, their system still performs well for almost all objects and almost all searchers. In their system, nodes have degree $O(\log^2 n)$. Naor and Wieder take a different approach. They build a constant-degree DHT (detailed in [9]), and then force each node to act for $O(\log n)$ others. Since the original degree was constant, the final degree is still a reasonable $O(\log n)$.

Section 4 gives similar techniques that can be used to "retrofit" Pastry and Tapestry with fault tolerance. These techniques may be useful even with constant overhead, and in Section 5 shows some simple experiments that suggests this.

### 1.2 The Main Idea

We address two problems. The first is how to build the data structures necessary for routing in the presence of faulty nodes. The second, is how to route securely. As mentioned above, we discuss Pastry [13] and Tapestry [17]. Both of these are closely related to the static object location system described by Plaxton, Rajaraman, and Richa in [11]. In these systems, routing is prefix routing based on the nodes IDs. This means that a node must store, for every prefix of its ID, a node with every one-digit extension of that prefix.

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

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

Section 3 looks at the difficulty of building a correct table when nodes send incorrect information. In systems like Chord, neighbors are fixed based on their ID, while in Pastry and Tapestry, nodes are given freedom to choose from among a set of choices. This flexibility gives better performance. However, both [15]

| Scheme | Redundacy | Extra Work fail-stop | bad-data | additional assumption? |
|---|---|---|---|---|
| Neighbor Table (Section 3) | $l$ | $O(l^2)$ | $O(l^2)$ | — |
| Routing I (Section 4.1) | $l$ | $O(l)$ | $O(l^2)$ | fraction bad nodes $< 1/c^2$ |
| Routing II (Section 4.2) | $l$ | $O(l^2)$ | $O(l^2)$ | — |

**Table 1.** Summary of paper results: All algorithms assume that $c^2 < b$, and that $l$ is $\Omega(\log n)$ and chosen sufficiently large that with high probability, one node in $l$ is good.

and [2] argue that this flexibility is harmful because it means that a node has a difficult time determining what information can be trusted.

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

Section 4 presents two routing techniques presented in this paper are similar in spirit to that of [14, 10] in that they route to a set of nodes at each level instead of just one. However, in this paper, these sets are not determined based on IDs, but on network distance. At each step, the algorithm queries the nodes in the current set to get the next set. If one node in the current set is good, the hope is that at least one node in the next step is also good. There is a subtlety here. If the the nodes in the current set return do not return the same set of nodes for the next step, the next set could be larger than the current set. This growth in the set size means that this technique could amount to flooding the network, which would be very inefficient.

However, if the algorithm eliminate some next set possibilities, there is a danger it will eliminate the only good possibilities. (And if failed nodes tend to return other failed nodes, the percentage of failed nodes may be higher than the percentage of failed nodes in the network as a whole.) The algorithm must ensure that it always keeps at least one good possibility for the next step. Our idea is to use the network distance from the query originator to determine which nodes are in the next set, since in our model, the adversary cannot manipulate ht network distance. For a small enough fraction of failing nodes, this succeeds with high probability when the sets are of size about $O(\log n)$. Section 4 presents two algorithms implementing this general idea. Both results rely on a restricted-growth assumption about the network, but these techniques may still perform well in other situations.

Table 1 summarizes our results.

### 1.3 Model

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

To explain our model, it helps to view the network as made up of two layers. The underlying layer can be trusted to deliver messages. The upper layer, or overlay layer, consists of the peer-to-peer nodes and the connections they maintain with each other (using the underlying layer for routing). This overlay is not trusted. These nodes can misbehave, but they cannot destroy or modify message on the wire, only messages that make go to the overlay layer. Figure 4 shows this distinction. Furthermore, this paper makes the following assumption about overlay nodes:

– Node IDs are assigned securely.
– Nodes fail, or become corrupted, independent of their location.
– The network must form a restricted-growth metric space. This essentially means that the number of nodes within distance $r$ is polynomial in $r$.
– Part of the trusted component is the ability to measure network distance(i.e., by pinging).
– For one of the secure routing algorithms (Section 4.1, we assume the fraction of bad nodes is small compared to the growth rate of the graph.

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

The major difference between our model and that of Castro *et al.* [2] is focused around our assumptions on the network topology. This paper requires a secure way of measuring network distance, supplemented with the assumption that nodes fail independently of their distance. While these requirements may seem limiting, note that changing distance measurements, particularly making them shorter, is actually quite difficult without interrupting the flow of messages at the lowest level—a capability for denial of service that is outside the scope of this paper (and of the Castro *et al.* solution).

Second, relaxation of the location independence requirement can lead to routing failure in regions with high local concentrations of bad nodes; in this circumstance, our techniques could be supplemented with any non-local technique (for example [2, 10, 14]).

## 2 Preliminaries

This section briefly describes the Tapestry neighbor table and routing algorithm (for the object location algorithm, see [7]). Note that the data structure presented here is almost identical to Pastry (and very similar to [11]), though the two systems differ markedly in how they handle object location. Discussing this difference is outside the scope of this paper.

A peer or node in Tapestry is assigned a random ID (all the IDs are the same length) with digits in some base $b$. The name space is chosen large enough so no two IDs will be the same. A message is routed toward a destination using prefix routing. That is, if the destination is 1234, the first routing hop goes to a node with an ID beginning with 1, the second hop goes to a node with an ID beginning with 12, and so on. To do this, for each prefix $\alpha$ of a node's ID, the node must keep a pointer to one node with each possible extension of that prefix. For example, for $\alpha = 12$, the node 1234 must keep a pointer to a node beginning with 120, 121, 122, etc.

When there is more than one node meeting a condition, the closest such node is chosen. As a result, links belonging to shorter prefixes in the neighbor tables tend to be shorter in network distance than the ones on higher levels, because there are more nodes meeting shorter prefixes. In the class of networks considered in this paper, these link lengths are geometrically increasing, so the distance traveled is proportional to the distance between the source and the destination, and is independent of the number of hops. Let $\mathcal{B}_r(A)$ be the ball of radius $r$ around $A$, or all the points within $r$ of $A$. Then the condition needed is

$$|\mathcal{B}_{2r}(A)| \leq c\,|\mathcal{B}_r(A)|\,, \qquad (1)$$

where $c$ the expansion constant of the network.

In words, this means that for a given node $A$, the number of nodes within $2r$ of $A$ is no more than a constant times the number of nodes within $r$ of $A$. Graphs that can be modeled as grids for some dimension $d$ meet this condition with $c = 2^d$. This condition probably does not hold on the Internet; however, this may still be a useful model in practice.

At some point in the routing, there may be no node matching a given ID, and the two systems differ in how they handle this case. Pastry keeps a list nodes with nearby IDs, called the leaf set. (In most cases, this list contains the nodes in the highest levels of the tree. It is useful in ways not discussed here, see [13].) When a hole is encountered in the routing table, routing proceeds using these nodes. Tapestry defines a substitute or surrogate node for the missing node and routes towards that node.

For the purposes of this paper, it is convenient to notice that each destination is the root of a tree containing all nodes in the network. Then, a route is a path from a leaf to the root of the tree. Relative to the root, each node has a level, where the level of a node is the length of the longest matching prefix with the root.[3] For simplicity, imagine that a level-$i$ node is also a level-$(i-1)$ node, a level-$(i-2)$ node, and so on. The longer the matching prefix, the higher up in the tree a node is, and a level-0 node is a leaf. Then every level-$i$ node takes as its parent the closest level-$(i+1)$ node, where closest refers to network distance (i.e. ping time). Sometimes nodes will have multiple parents. The $k$ parents of a level-$i$ node are the $k$ closest level-$(i+1)$ nodes to be its parents.

A path through the network is then a path from a leaf in this tree to the root. For example, the ID 1234, the root of the tree is the node with ID 1234, its children are the nodes with IDs beginning with 123, and their children are the nodes beginning with ID 12, and so on. Each node in the network then

[3] In Pastry, using the leaf sets bypasses some of these levels, however, in almost all cases, the number of levels bypassed this way is small.

participates in $n$ trees, but in only needs to store $b$ parents at each level in the tree (see [11, 13, 7]). This means that its total storage is $b \log_b n$. For simplicity and ease of exposition, the rest of this paper imagines that there is only one tree.

A recent paper [7] shows how to build these neighbor tables in a distributed way. Here, we improve on that algorithm by making it more tolerant to faults. First, let us state a lemma from that paper. Suppose a ball around $A$ of radius $r$ contains $k$ nodes at level-$i$. The lemma bounds the probability that a larger ball around $A$ (of size $3r$) contains $k$ level-$i$ nodes that also satisfy predicate $p(X)$ where nodes satisfy $p(X)$ independently with probability $f$. The following lemma is proved, though not in exactly this form, in [7].

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

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

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

## 3 Building the Neighbor Table in the Presence of Faults

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

Viewing the algorithm as a black box would require running it once for every prefix. In fact, it can also be used more cleverly. By choosing the appropriate starting point, in one pass it finds one entry at every level. Further, [7] showed that by choosing parameters carefully, we can use this technique to fill the entire table.

### 3.1 The Fragile Algorithm

Recall that nodes are divided into levels of geometrically decreasing size, such that the $i$th level contains roughly $n/b^i$ nodes. Further, recall that a level-$i$ node chooses as its parent (or parents) the closest level-$(i+1)$ node. (This arrangement was motivated by [1, 11, 18] which used that to get paths through the network that not only had few hops, but also short network distance.) Figure 5 shows the algorithm of [7]. Section 3.2 will explain how a slight modification in the data structures used by this algorithm makes the algorithm much more robust.

The algorithm starts with a list of the $k$ closest nodes at *maxLevel*. We call this list the **RootSet**, since in the fragile version of the algorithm, this set

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

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

**Fig. 5.** Finding the Nearest Neighbor: This algorithm operates with respect to the tree defined by the **RootSet**. For more detail, see text.

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

## 3.2 The Robust Algorithm

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

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

With $l$ parents, and a failure probability of $f$, the probability that all the parents fail is $f^l$ (here, failure means the node should return $B$ but either does not reply, or returns a list without $B$). If $l = O(\log n)$ and $f$ is constant, then the probability of a failure is an inverse polynomial in $n$. (Unless the root is assumed to be good, the **RootSet** must also be of size $l$. If this is not given to the algorithm, we can use the fault tolerant routing described in Section 4 to find a set of nodes at the right level.)

In order for this argument to hold, the algorithm must query all of $B$'s parents, so clearly $k$ must be large enough that the algorithm will query all of $B$'s parents. The following argument shows that this can be done with $k$ still

$O(\log n)$ (so long as $l = O(\log n)$). The idea is that any nearby node has its parents nearby as well, for the correct definitions of nearby.

**Lemma 2.** *For $k > bl$, with $l = \Omega(\log n)$, with high probability, the $k$ closest level-$(i+1)$ nodes contain all the parents of the closest $k$ level-$i$ nodes.*

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

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

## 4 Routing in a Faulty Network

In [2], the authors present a routing technique for dealing with hostile networks. Their idea is to use $r$ different paths to the root, and then they argue that if a fraction $f$ of the nodes are corrupted, then the probability a message reaches the destination along a particular path is $(1-f)^{1+\log_b n}$. The probability that all $r$ of the paths fail is $((1-f)^{1+\log_b n})^r$. Asymptotically, this is a rather low probability of success. In fact, using their formulation, the probability of failure is more than $(1-(1-f)^{1+\log_b n})^r$, which is approximately $\exp(-rn^{\frac{\ln(1-f)}{\ln b}})$. If the desired failure probability is constant, then $r$ must be a polynomial in $n$.

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

If an $i$th hop node returns nodes that look like good $(i+1)$st hops but are not, then the initiating node does not know which of the nodes are good. Consider the case where the nodes are malicious but of limited power and want to pass messages on to a particular subnet (perhaps their own, because they control it, or perhaps another to get rid of traffic). Picking from among the returned nodes at random may also be a problem, since the misconfigured nodes may be more likely to return other misconfigured nodes, while the correct nodes may return misconfigured nodes with probability proportional to the fraction of misconfigured nodes. Both of the techniques of this section attempt to deal with this case by using network distance information to pick from this set of returned nodes ones that are reasonably likely to be good.

Assume that the fraction of faulty nodes is $f$. Both of our techniques require each node to store not one neighbor in each entry in the routing table, but $l = O(\log n)$. In the tree terminology, this means each node stores not one

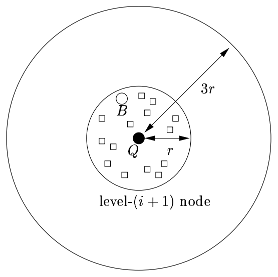

**Fig. 6.** The parents of $B$ lie within the big circle. The squares represent some of the level-$(i+1)$ nodes that $B$ could choose as parents.

parent, but $l$ parents, so the structure is no longer a tree, but the parent and child terminology still applies.

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

## 4.1 Routing Technique I

The technique is in some sense a sort of reverse of the nearest neighbor algorithm described in the previous section. The algorithm works as follows.

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

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

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

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

*Proof.* See Figure 6. Suppose $B$ is a level-$i$ node within distance $r$ of $Q$. Note that $B$ has $l$ potential parents within $r$ of $Q$. By the triangle inequality, all these nodes are within $2r$ of $B$. If $B$ chooses different parents than the nodes within the smaller ball, it could only be because they were closer, so all of $B$'s parents are within $2r$ of $B$. But this means they are within $3r$ of $Q$. $\square$

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

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

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

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

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

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

## 4.2 Routing Technique II

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

At every step, this algorithm ensures that it knows the closest $l$ level-$i$ nodes. In Section 4.1, the algorithm did not guarantee that it had the "closest" $l$ nodes. Knowing that they are the closest means they are determined by the structure of the network and not by the misbehavior of the bad nodes, so the probability of failures is independent (given our network assumptions) among these $l$ nodes. The algorithm works as follows:

1. From the level-$i$ nodes, pick all the level-$(i+2)$ nodes. In Lemma 4, we show that $l = O(\log n)$ is big enough such that with high probability, at least one of the level-$i$ nodes is good level-$(i+2)$ node.
2. Get the children of these nodes.
3. Pick the closest $l$ of these children to be the set of level-$(i+1)$ nodes.

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

**Lemma 4.** *For $l \geq \log(1/\epsilon)\frac{b^2}{1-f}$, the probability than none of the level-i nodes are good level-$(i+2)$ nodes is less than $\epsilon$.*

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

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

**Lemma 5.** *Suppose $A$ is a level-$(i+2)$ node and is among the closest $l$ level-$i$ nodes to $B$. Then for $l = O(\log n)$, with high probability, the closest $l$ level-$(i+1)$ nodes to $B$ point to $A$.*

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

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

## 5 Experiments

We implemented the nearest neighbor (Section 3) algorithm and the routing algorithm of Section 4.1. The simulation used 50,000 nodes, using a base of 10 (this means the number of steps to the root was five or six.) The underlying topology used was a grid, where the overlay points were chosen at random.

The failed nodes in both cases only return information about other failed nodes. This is worse that the fail-stop model, since here nodes are actually getting bad data, but is not the worst that bad nodes could do.

For varying fractions of bad nodes, we ran the nearest neighbor algorithm, and calculated the number of times that algorithm gave the incorrect answer because of the failed nodes.[4] See Figure 7. Notice that when half the nodes are failed, the number of incorrect (i.e., not closest) nodes actually decreases; this is because return only other failed nodes does not cause problems if the the end answer is also a failed node.

[4] The algorithm implemented here is an improved version of the algorithm described in [7] that follows the same general outline but changes $k$ during the course of the algorithm. For details, see [6].

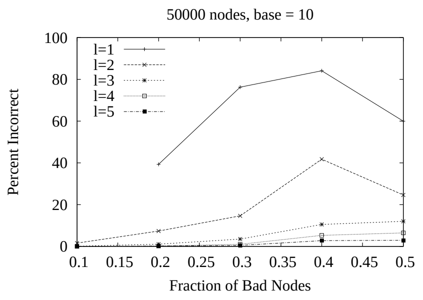

**Fig. 7.** Percentage of incorrect entries over 5,000 trials in a network of 50,000 nodes. Even a small amount of redundancy (shown here as $l$) significantly reduces the number of incorrect entries.

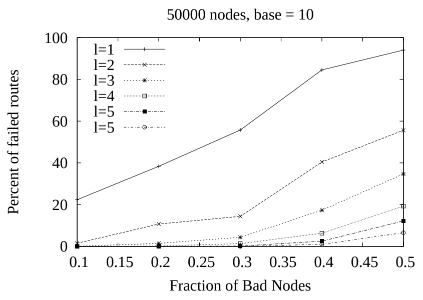

**Fig. 8.** Percentage of routes that fail to reach the root when the algorithm of Section 4.1 is used. Notice that a small amount of redundancy (shown here as $l$) helps tremendously.

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

Figure 8 shows that chance of reaching the root improves a great deal with only a little additional overhead. Note that Tapestry already stores two backups for every entry, so $l = 3$ requires no additional overhead. (And the nearest neighbor algorithm uses these backups.)

## 6 Conclusion

Tolerating misbehaving components is a requirement for any large-scale system. In this paper, we took a step toward achieving this goal for peer-to-peer overlay networks—by harvesting redundancy. We showed how to build routing tables that take advantage of locality in the network even in the presence of faulty nodes. We also presented a technique for fault-tolerant routing that gives a high probability of success with small overhead. Although we applied these techniques to the specific instances of Tapestry and Pastry, these techniques appear to be generally applicable and could enhance other systems.

## Acknowledgments

Thanks are due to Satish Rao, Sean Rhea, Jeremy Stribling, and the OceanStore group for their comments on this paper and many helpful discussions.

## References

1. M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, 2002.

2. Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Desgin and Implementation*, pages 299–314, 2002.
3. John Douceur. The Sybil attack. In *Proceedings of IPTPS*, pages 251–260, 2002.
4. Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of Symposium on Discrete Algorithms*, 2002.
5. Nicholas J.A. Harvey, Marvin Theimer Michael B. Jones, Stefan Sarouiu, and Alec Wolman. Skipnet: A peer-to-peer overlay network. In *Proceedings of the fourth USENIX Symposium on Internet Technologies and Systems*, 2003.
6. Kirsten Hildrum, John Kubiatowicz, and Satish Rao. Improved bounds on finding nearest neighbors in growth-restricted metrics. `http://www.cs.berkeley.edu/~hildrum/nn.ps`.
7. Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, 2002.
8. Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, 2002.
9. Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 50–59, 2003.
10. Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Second International Workshop on Peer-to-Peer Systems*, 2003.
11. C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th Annual Symp. on Parallel Algorithms and Architectures*, pages 311–320, 1997.
12. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, pages 161–172, 2001.
13. Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, 2001.
14. Jared Saia, Amos Fiat, Steve Gribble, Anna R. Karlin, and Stefan Saroiu. Dynamically fault-tolerant content addressable networks. In *First International Workshop on Peer-to-Peer Systems*, 2002.
15. Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.
16. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, pages 149–160, 2001.
17. Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.
18. Ben Y. Zhao, Anthony Joseph, and John Kubiatowicz. Locality-aware mechanisms for large-scale networks. In *Proc. of Workshop on Future Directions in Distributed Comp.*, 2002.