Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems

Antony Rowstron[1] and Peter Druschel[2]*

[1] Microsoft Research Ltd, St. George House,
1 Guildhall Street, Cambridge, CB2 3NH, UK.
antr@microsoft.com
[2] Rice University MS-132, 6100 Main Street,
Houston, TX 77005-1892, USA.
druschel@rice.cs.edu
PRELIMINARY DRAFT

Abstract. This paper presents the design and evaluation of Pastry, a scalable, distributed object location and routing scheme for wide-area peer-to-peer applications. Pastry performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet. It can be used to support a wide range of peer-to-peer applications like global data storage, global data sharing, and naming.

An insert operation in Pastry stores an object at a user-defined number of diverse nodes within the Pastry network. A lookup operation reliably retrieves a copy of the requested object if one exists. Moreover, a lookup is usually routed to the node nearest the client issuing the lookup (by some measure of proximity), among the nodes storing the requested object. Pastry is completely decentralized, scalable, and self-configuring; it automatically adapts to the arrival, departure and failure of nodes.

Experimental results obtained with a prototype implementation on a simulated network of up to 100,000 nodes confirm Pastry's scalability, its ability to self-configure and adapt to node failures, and its good network locality properties.

1 Introduction

Peer-to-peer Internet applications for global file sharing like Napster, Gnutella and FreeNet [1–4] have recently gained popularity. Several research projects aim at constructing other types of peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [5, 4]. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities, and all communication is symmetric.

One of the key problems in large-scale peer-to-peer applications is to provide efficient algorithms for application-level routing and location of content within the network. Currently, each peer-to-peer application uses it own approach to this problem. For instance, Napster locates content using a centralized Website; Gnutella relies on

* Work done in part while visiting Microsoft Research, Cambridge, UK.

broadcast to locate content; and FreeNet uses randomized content routing that gains scalability and a degree of anonymity at the expense of reliable content location.

This paper presents Pastry, a generic, decentralized peer-to-peer content location and routing system for very large, self-configuring overlay networks of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably locates a nearby copy of the requested content. Pastry can be be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, and naming systems.

An *insert* operation in Pastry stores an object at a user-defined number of diverse nodes within the Pastry network. A *lookup* operation reliably retrieves a copy of the requested object if one exists. Usually, the object is retrieved from the live node "nearest" the client issuing the lookup, among the nodes storing the object. Proximity is measured here in terms of a scalar, application defined metric, such as the number of network hops or network delay. Depending on the application, an object can hold content or a locator (address) of the associated content.

Pastry is completely decentralized, is self-configuring, it automatically adapts to the arrival, departure and failure of nodes, and it is scalable. The number of nodes traversed, as well as the number of messages exchanged while routing a client request is at most logarithmic in the total number of nodes in the system.

Although Pastry is intended as a generic building block for peer-to-peer applications, it was designed in the context of the PAST project, an Internet based, peer-to-peer global storage utility. PAST aims to provide strong persistence, high availability, scalability, content privacy and anonymity of clients and storage providers. This paper focuses on Pastry, PAST's content location and routing system.

The rest of this paper is organized as follows. Section 2 presents the design of Pastry. Experimental results with a prototype implementation of Pastry are presented in Section 3. Related work is discussed in Section 4 and Section 5 concludes.

2 Design of Pastry

A Pastry system is a self-organizing overlay network of nodes, where each node routes client requests and is capable of storing application-specific objects. Any computer that is connected to the Internet and runs the Pastry node software can act as a Pastry node, subject only to application-specific security policies.

Inserted objects are replicated across multiple nodes. The system ensures, with high probability, that the set of nodes over which an object is replicated is diverse in terms of geographic location, ownership, administrative entity, network connectivity, rule of law and so forth.

Each node in the Pastry peer-to-peer overlay network is assigned a 128-bit node identifier (nodeId). The nodeId is used to indicate a node's position in a circular namespace, which ranges from 0 to $2^{128} - 1$. This nodeId is drawn randomly when a node joins the system. It is assumed that the nodeId is generated by a high-quality uniform random number generator or a secure hash function, so that the set of existing node identifier is uniformly distributed in the 128-bit namespace.

The fundamental capability Pastry provides is to efficiently route messages among the nodes in the system. Specifically, given a destination id (destId) of at least 128 bits, Pastry routes an associated message to the node whose nodeId is numerically closest to the 128 most significant bits of the destId associated with the message, among all live nodes.

Furthermore, it is assumed that each object is assigned an object id (objId) that is at least 128 bits long. The set of existing objIds is further assumed to be uniformly distributed. To insert an object, a client asks Pastry to route a message to the node whose nodeId is numerically closest to the 128 most significant bits of the objId; that node then stores the object. To look up an object, a client similarly sends a message using the objId as the destId; the receiving node responds with the requested object.

To achieve high availability and/or load balancing, an object is stored on the k nodes whose nodeIds are numerically closest to the objId. The object is then available as long as one of the k nodes is live and reachable (in the Internet) from a client. Furthermore, Pastry ensures, with high probability, that a lookup message is routed to one of the k nodes that is near the client. This provides for load balancing, reduced network load and low client response time.

In the context of the PAST global storage utility, for instance, the objIds are formed from a secure hash (SHA-1) of an object's name, content, and the object owner's identifier. An object is stored at the k nodes with nodeIds closest to the objId. As long as one of the k copies is alive, Pastry guarantees that a lookup message will be routed to that node. Moreover, a lookup message in Pastry usually first reaches the node nearest the client, among the live subset of the k nodes.

2.1 Pastry nodeIds

A nodeId is sub-divided into a sequence of *levels*, where each level specifies a *domain*, represented by b contiguous bits in the nodeId[1]. The bits at positions $b*l$ to $b*(l+1)-1$ specify the domain at level l. That is, the most significant b bits of the nodeId indicate the node's domain at level 0, and so on. There are 2^b domains at each level, numbered from 0 to $2^b - 1$.

Pastry routes messages to the node whose nodeId is numerically closest to a given destId. This is accomplished as follows. At each routing step, a message whose destId matches the local node's nodeId up to level l is forwarded to a node whose nodeId matches the destId up to at least $l + 1$. For this purpose, each node maintains some routing state, which we describe next.

2.2 Pastry node state

Each Pastry node maintains a *routing table*, a *neighborhood set* and a *namespace set*. We begin with a description of the routing table. For each level l, the routing table contains the IP addresses of $2^b - 1$ nodes that have the same nodeId prefix as the local node up to level $l - 1$, but differ at level l. Each of these nodes is a *representative* of a different domain at level l.

[1] Typically a value of 3 or 4 would be used for b.

In principle, any node whose nodeId matches the local node's nodeId up to level $l-1$ and whose domain at level l equals d can serve as a representative for domain d. In practice, among all nodes with the correct nodeId prefix, the node that is closest to the present node in the network is chosen as the representative. As will be shown in Section 2.4, this ensures that message routing in Pastry exhibits good network locality.

The choice of b involves a tradeoff between the size of the populated portion of the routing table (approximately $\lceil log_{2^b} N \rceil \times (2^b - 1)$, where N is the total number of existing Pastry nodes) and the maximum number of hops required to route between any pair of nodes ($\lceil log_{2^b} N \rceil$) [2]. With a value of $b = 4$ and with as many as 10^{12} nodes, the routing table contains only approximately 150 entries and in the worst-case a message is routed through 10 nodes.

The neighborhood set M contains the nodeIds and IP addresses of the $|M|$ nodes that are closest (according the proximity metric) to the local node. The neighborhood set is not normally used in routing messages; its purpose will become clear in Section 2.5.

The namespace set L contains the nodeIds and IP addresses of the $|L|$ existing nodes whose nodeIds are numerically closest and centered around the local node's nodeId. The namespace set is used during the message routing, as described below. The set is also used during object insertion, where k replicas of the inserted object are stored on a subset of the namespace set. Typical values for $|L|$ and $|M|$ are 2^b and 2×2^b, respectively.

How the various tables of a Pastry node are initialized and maintained is the subject of Section 2.5. Figure 1 depicts the state of a hypothetical Pastry node with the nodeId 10233102 (base 4), in a system that uses 16 bit nodeIds and a value of $b = 2$.

2.3 Routing

The routing procedure is shown in pseudocode form below. It is executed whenever a message with destId D arrives at a node with nodeId A. We begin by defining some notation.

R_l^i: the entry in the routing table R for domain i, $0 \leq i < 2^b$ at level l, $0 \leq l < \lfloor 128/b \rfloor$.

M_i: the entry in the neighborhood table M, representing the i-th closest node, $0 \leq i < |M|$.

L_i: the i-th closest nodeId in the namespace table L, $-\lfloor |L|/2 \rfloor \leq i \leq \lfloor |L|/2 \rfloor$, where negative/positive indices indicate nodeIds smaller/larger than the present nodeId, respectively.

D_l: the domain of destId D at level l.

$shl(A, B)$: the length of the prefix shared among A and B, in levels.

```
(1) if ($L_{-\lfloor |L|/2 \rfloor} \leq D \leq L_{\lfloor |L|/2 \rfloor}$) {
(2)     // $D$ is within range of our namespace set
(3)     forward to $L_i$, s.th. $|D - L_i|$ is minimal;
(4) } else {
(5)     // use the routing table
(6)     Let $l = shl(D, A)$;
```

[2] We assume throughout this paper that nodeIds are uniformly distributed.

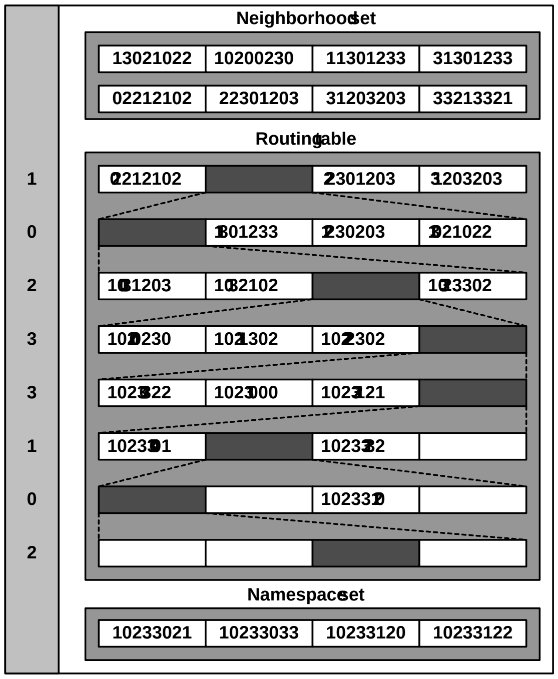

Fig. 1. State of a hypothetical Pastry node with nodeId 10233102, $b = 2$. All numbers are in base 4. The top row of the routing table represents level zero.

```
(7)        if ($R_l^{D_l} \neq null$) {
(8)             forward to $R_l^{D_l}$;
(9)        }
(10)       else {
(11)            // rare case
(12)            forward to $T \in L \cup R \cup M$, s.th.
(13)                 $shl(T, D) \geq l$,
(14)                 $|T - D| < |A - D|$
(15)       }
(16) }
```

Given a message, the node first checks to see if the destId falls in the range of nodeIds covered by its namespace set (line 1). If so, the message is forwarded directly to the destination node, namely the node in the namespace set whose nodeId is closest to the destId (possibly the present node) (line 3).

If the destId is not covered by the namespace set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the destId by at least one more level (lines 6–8). In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable (line 11–14), in which case the message is forwarded to a node that shares a prefix with the destId at least as long as the local node, and is numerically closer to the destId than the current nodeId. It follows from the properties of the routing table and namespace set that such a node must always exist, unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeIds have failed simultaneously.

This simple routing procedure always converges, because each step takes the messages to a node that either (1) shares a longer prefix with the destId than the local node, or (2) shares as long a prefix with, but is numerically closer to the destId than the local node.

Routing performance It can be shown that a message reaches its destination in no more than $\lceil log_{2^b} N \rceil$ steps in the common case, where N is the total number of existing Pastry nodes. Briefly, consider the three cases in the routing procedure. If a message is forwarded using the routing table (lines 6–8), then the set of nodes containing the destination node is reduced by a factor of 2^b in each step, which means the destination is reached in $\lceil log_{2^b} N \rceil$ steps. If the destId is within range of the namespace set (lines 2–3), then the destination node is at most one hop away.

The third case arises when the destId is not covered by the namespace set (i.e., it is still more than one hop away from the destination), but there is no routing table entry. In the absence of node failures, this means that a node in the corresponding domain does not exist (lines 11–14). Due to the uniform distribution of nodeIds this case is unlikely, provided $|L|$ is sufficiently large. For instance, with $|L| = 2^b$ and $|L| = 2 \times 2^b$ it occurs in less than 2% and 0.4% of all cases, respectively. When it happens, at most one additional routing step results in virtually all cases.

In the event of many simultaneous node failures, the number of routing steps required may be at worst linear in N, while the nodes are updating their state. In practice, routing performance degrades gracefully, as we will show experimentally in Section 3.1. Ultimately, a node may become unreachable when at least $\lfloor |L|/2 \rfloor$ nodes with consecutive nodeIds fail simultaneously. However, due to the expected diversity of nodes with adjacent nodeIds, and with a reasonable choice for $|L|$ (e.g. 2^b), the probability of this event is very low.

2.4 Network locality

In the previous section, we discussed Pastry's convergence and the expected number of routing hops. This section focuses on another aspect of Pastry's routing performance, namely its properties with respect to network locality. We will show that (1) the route chosen for a message is likely to be "good" in terms of network proximity, and (2) if an object is stored at k consecutive nodes, a query message for the object is likely to first reach a node near the client, among the k nodes.

Pastry's notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops, the network delay, or a combination of these and other factors. All that Pastry assumes is that the application provides a function that allows each Pastry node to determine the "distance" of a node with a given IP address to itself. A node with a lower distance value is assumed to be more desirable. An application implements this function depending on its choice of a proximity metric, using network services like traceroute, ping, or Internet subnet maps and appropriate caching and approximation techniques to minimize overhead.

Route locality Recall that the representative for each domain and level in the routing table is chosen to be the node closest in the network to the present node, among all nodes with the given nodeId prefix. As a result, in each routing step, a message is forwarded to the closest node with a nodeId that shares a longer common prefix or is numerically closer to the destId than the local node. That is, each step moves the message closer to the destination in the namespace, while travelling the least possible distance in the network.

Since only local information is used, Pastry minimizes the distance of the next routing step with no sense of direction. This procedure clearly does not guarantee that the shortest path from source to destination is chosen; however, it does give rise to reasonably good routes. Two facts are relevant to this statement. First, given a message was routed from node A to node B at distance d from A, the message cannot subsequently be routed to a node with a distance of less than d from A. This follows directly from the routing procedure, assuming accurate routing tables.

Second, the expected distance traveled by a messages during each successive routing step is exponentially increasing. To see this, observe that a representative in the routing table at level l is chosen from a set of nodes of size $N/2^{bl}$. That is, the representatives at successive levels are chosen from an exponentially decreasing number of nodes. Given the random and uniform distribution of nodeIds in the network, this means that the expected distance of the closest representative at each successive level is exponentially increasing.

Jointly, these two facts imply that although it cannot be guaranteed that the distance of a message from its source increases monotonically at each step, a message tends to make larger and larger strides with no possibility of returning to a node within d_i of any node i encountered on the route, where d_i is the distance of the routing step taken away from node i. Therefore, the message has nowhere to go but towards its destination. Figure 2 illustrates this effect.

Locating the nearest replica We now turn to our second claim; namely, among k nodes with adjacent nodeIds that store an object, a client query will likely be routed first to a node that is near the client. Observe that due to the random assignment of nodeIds, nodes with adjacent nodeIds are likely to be widely dispersed in the network. Thus, it is important to direct a lookup query towards a replica that is located relatively near the client.

Recall that Pastry routes messages towards the node with the nodeId closest to the destId, while attempting to travel the smallest possible distance in each step. Therefore, among the k nodes storing an object, a query message tends to first reach a node near the client. Of course, this process only approximates routing to the nearest replica. Firstly, as discussed above, Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no notion of direction.

Secondly, since Pastry routes primarily based on nodeId prefixes, it sometimes misses nearby replicas stored on nodes with a different prefix than the object. In the worst case, $k/2 - 1$ of the replicas are stored on nodes whose nodeIds differ from the

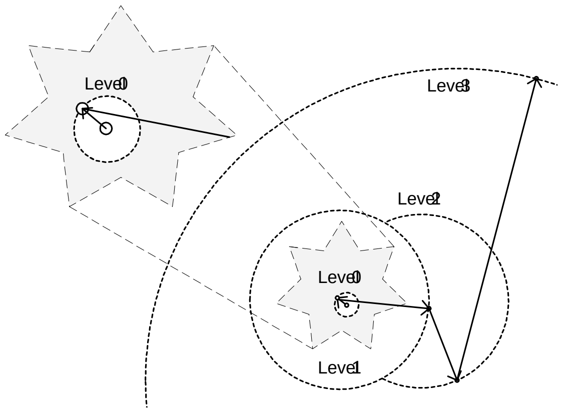

Fig. 2. Trajectory of a typical message in the Pastry network, based on experimental data. The message cannot re-enter the circles representing the distance of each of its routing steps away from intermediate nodes. Although the message may partly "turn back" during its initial steps, the exponentially increasing distances traveled in each step cause it to move toward its destination quickly.

objId in their domain at level zero. As a result, Pastry will first route towards the nearest among the $k/2 + 1$ remaining replicas. Despite this anomaly, results presented in Section 3.3 show that this and similar cases occur infrequently enough that Pastry is able to locate the nearest replica in approximately 60%, and one of the two nearest replicas in approximately 80% of all queries.

Moreover, we are exploring heuristics to overcome the prefix mismatch issue described above. One very simple heuristic we have studied is based on estimating the density of nodeIds in the namespace using local information. Based on this estimation, the heuristic detects when a message approaches the replica set of an object, and then switches to numerically nearest address based routing to located the nearest replica. Our results show that this heuristic allows Pastry to locate the nearest object in over 75%, and one of the two nearest replicas in over 91% of all queries, at the expense of a slight increase in the average number of hops taken.

2.5 Self-configuration and adaptation

In this section, we discuss how Pastry deals with changes in node membership. In particular, we describe the protocols handling the arrival and departure of nodes in the Pastry network. Throughout this discussion, we assume that the proximity space defined by the chosen proximity metric is euclidian; that is, the triangulation inequality holds for distances among Pastry nodes. If this assumption does not hold, routing correctness is unaffected; however, the locality properties of Pastry routes may suffer.

Node arrival When a new node arrives, it needs to initialize its tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby (in the network) Pastry node A that is already part of the system. Such a node can be detected automatically, for instance, using “expanding ring” IP multicast, or obtained by the system administrator through outside channels.

The new node draws a random nodeId X and then asks A to route a special “join” message with the destination id equal to X. Like any message, the join will be routed to the existing node Z whose id is numerically closest to X.

In response to receiving the “join” request, nodes A, Z, and all nodes encountered on the path from A to Z send their state to X. The new node X inspects this information, requests state from additional nodes, and then initializes its state, using a procedure describe below. Finally, X informs any nodes that need to be aware of its arrival. We will show that this procedure ensures that X initializes its state with appropriate values, and that the state of all other interested nodes are modified appropriately.

First, consider the neighborhood set. Since node A is assumed to be close to the new node X, A’s neighborhood set is a close approximation of X’s neighborhood set, and can therefore be used to initialize the latter. Second, since Z has the closest existing nodeId to X, its namespace set is the basis of X’s namespace set. Furthermore, Z provides X with information about object replicas stored within its namespace set, allowing X to properly forward lookup messages for objIds within the range of the namespace set.

Next, we consider the routing table, starting at level zero. We consider the most general case, where the nodeIds of A and X share no common prefix. Since all nodes share the same level zero domains, the representatives at this level only depend on a node’s location. Let A_i denote node A’s row of the routing table at level i. Since A is assumed to be close to X, A_0 closely approximates the optimal values for X_0. Other levels of A’s routing table are of no use to X, since A’s and X’s ids share no common prefix.

However, appropriate values for X_1 can be taken from B_1, where B is the first node encountered along the route from A to Z. To see this, observe that B_1 mentions the same domains as X_1 because X and B share the same prefix at level 0. Intuitively, it would appear that the choice of representatives in B_1 is not appropriate, since these nodes are close to B, but not necessarily to X.

To see why this is not so, recall that the representatives at each successive level are chosen from an exponentially decreasing set size. Therefore, the expected distance from B of its B_1 representatives is much larger than the expected distance traveled from node A to B. As a result, B_1 is still a good approximation for X_1. This same argument applies for each successive level and routing step, as depicted in Figure 3.

After X has initialized its routing table in this fashion, it has enough information to participate in the Pastry network. However, at this point its routing table and neighborhood set only approximate the closest nodes (within each domain). The quality of this data must be improved to avoid cascading errors that could eventually lead to poor route locality. For this purpose, there is a second stage in which X requests the state from each of the nodes in its routing table and neighborhood set. It then compares the distance of corresponding representatives found in those nodes’ routine tables and neighborhood sets, respectively, and updates its own state with any closer nodes it finds.

Intuitively, a look at Figure 3 illuminates why incorporating the state of nodes mentioned in the routing and neighborhood tables from stage one provides good representa-

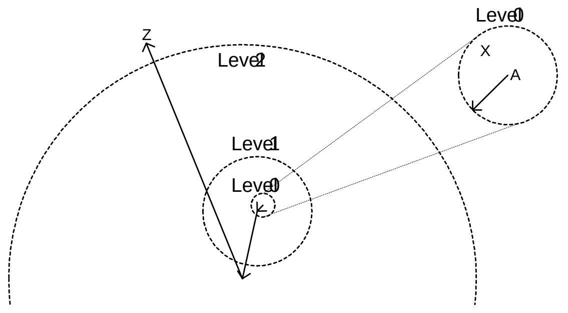

Fig. 3. Routing step distance versus distance of the representatives at each level (based on experimental data). The circles around the n-th node along the route from A to Z indicate the average distance of the node's representatives at level n. Note that X lies within each circle.

tives for X. The circles show the average distance of the representative from each node along the route, corresponding to the levels in the routing table. Observe that X lies within each circle, albeit off-center. In the second stage, X obtains the state from the representatives discovered in stage one, which are located at an average distance equal to the perimeter of each respective circle. These states must include representatives that are appropriate for X, but were not discovered by X in stage one, due to its off-center location.

Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, namespace set, and routing table. Those nodes in turn update their own state based on the information received. Experimental results in Section 3.2 show that this procedure initializes a node's state correctly and that it updates the state of relevant nodes appropriately. The total cost of joining a node, in the number of RPCs exchanged, is $O(log_{2^b} N)$. The constant is about 3×2^b (omitting the second stage reduces the constant to 2×2^b). For 100,000 nodes the largest message size is approximately 1 KByte.

Pastry uses an optimistic approach to controlling concurrent node arrivals and departures. Since the arrival/departure of a node affects only a small number of existing nodes in the system, contention is rare and an optimistic approach is appropriate. Briefly, whenever a node A provides state information to a node B, it attaches a timestamp to the message. B adjusts its own state based on this information and eventually sends an update message to A (e.g., notifying A of its arrival). B attaches the original timestamp, which allows A to check if its state has since changed. In the event that its state has changed, it responds with its updated state and B restarts its operation.

Node departure Nodes in the Pastry network may fail or depart without warning. In this section, we discuss how the Pastry network handles such node departures.

Node failures are detected when another node attempts to contact a node in its routing table or namespace set and there is no response. As explained in Section 2.3, such an event does not normally delay the routing of a message, since the message can be forwarded to another node. However, a replacement must be found to preserve the integrity of the routing table and namespace set.

To replace a failed node in the namespace set, a node contacts the live node with the largest index on the side of the failed node, and asks that node for its namespace table. For instance, if L_i failed for $\lfloor |L|/2 \rfloor < i < 0$, it requests the namespace table from $L_{-\lfloor |L|/2 \rfloor}$. Let the received namespace set be L'. This set partly overlaps the present node's namespace set L, and it contains nodes with nearby ids not presently in L. Among these new nodes, the appropriate one is then chosen to insert into L, verifying that the node is actually alive by contacting it. This procedure guarantees that each node lazily repairs its namespace set unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeIds fail. Due to the diversity of nodes with adjacent nodeIds, such a failure is very unlikely even for modest values of $|L|$.

To repair a failed representative R_l^d, a node contacts first another representative $R_l^i, i \neq d$ at the same level, and asks for its value for R_l^d. In the event that none of the representatives at level l have a pointer to a live representative in domain d, the node next contacts a representative $R_{l+1}^i, i \neq d$, thereby casting a wider net to include representatives that are likely to be further away in the network. This procedure eventually finds a representative if one exists.

The neighborhood set is not normally used in the routing of messages, yet it is important to keep it current, since the set is needed when a joining node requests it. For this purpose, a node attempts to contact each member of the neighborhood set periodically (e.g. once an hour) to see if it is still alive[3]. If a member is not responding, the node asks other members for their neighborhood tables, checks the distance of each of the newly discovered nodes, and updates it own neighborhood set accordingly.

Experimental results in Section 3.2 demonstrate Pastry's effectiveness in repairing the node state in the presences of node failures, and quantify the cost of this repair in terms of the number of messages exchanged.

3 Experimental results

In this section, we present experimental results obtained with a prototype implementation of Pastry. The Pastry node software was implemented in Java. To be able to perform experiments with large networks of Pastry nodes, we also implemented a network simulation environment, where up to 100,000 Pastry nodes can run over a simulated network.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 2GBytes of main memory, running True64 UNIX, version 4.0F. The Pastry node software was implemented in Java and executed using Compaq's Java 2 SDK, version 1.2.2-6 and the Compaq FastVM, version 1.2.2-4.

The Pastry nodes normally use Java remote object invocation (RMI) to communicate with each other. However, in all experiments reported in this paper, the Pastry nodes were configured to run in a single Java VM. This is largely transparent to the Pastry implementation—the Java runtime system automatically reduces communication among the Pastry nodes to local object invocations.

The simulated network environment maintains distance information between the Pastry nodes. Each Pastry node is assigned a location in a plane; coordinates in the

[3] An inactive node should do the same for its namespace set and routing table.

plane are randomly assigned in the range $[0, 1000]$. As Pastry is an overlay network, we assume that the underlying networking infrastructure used by Pastry provides total connectivity between all the nodes.

To drive the Pastry system, fictitious objects are inserted and retrieved. The objIds are generated by computing SHA-1 secure hashcodes of URLs collected by a Web crawler from several major University Web sites in the US. In all experiments, 5 replicas were stored for each inserted object ($k = 5$).

3.1 Routing performance

The first experiment shows the number of routing hops as a function of the size of the Pastry network. We vary the number of Pastry nodes from 1,000 to 100,000 in a network where $b = 4$, $|L| = 16$, $|M| = 32$. 100,000 objects are inserted and then each object is retrieved from a different, randomly chosen starting node. Figure 4 show the results. “Standard” is the normal Pastry routing procedure, “Estimation” is the routing procedure augmented with the heuristic mentioned in Section 2.4. “Log N” shows the value $log_{2^b} N$ and is included for comparison. ($\lceil log_{2^b} N \rceil$ is the expected maximum number of hops required to route in a network containing N nodes). The results show that the number of route hops scale with the size of the network as expected. Moreover, the cost of the routing heuristic to improve the location of nearby replicas, in terms of the average number of route hops, is insignificant. Both “Standard” and “Estimation” require less than $log_{2^b} N$ hops due to the namespace set. Occasionally, a hop is saved because the destination node lies in the namespace set, and therefore, is routed to directly, rather than requiring an extra hop.

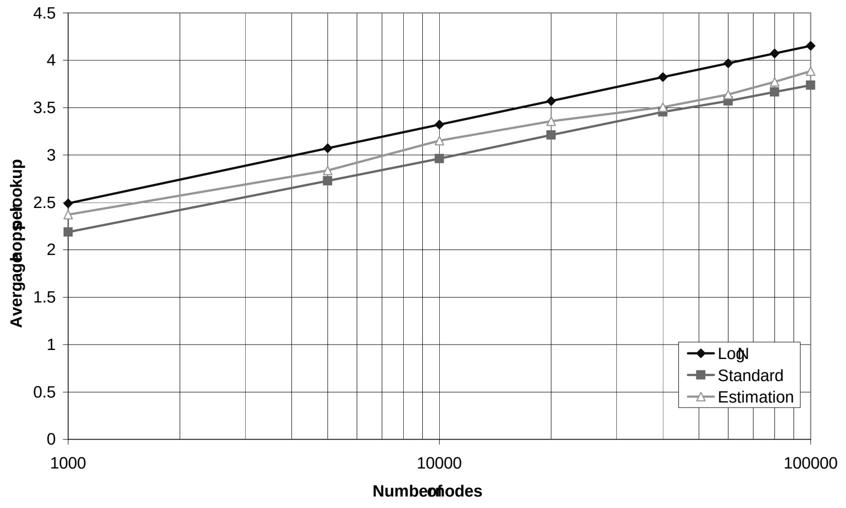

Fig. 4. Number of routing hops versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$ and 100,000 objects.

The second experiment compares the distance a message travels using Pastry with that of a fictitious routing scheme that maintains complete routing tables. Here, distance traveled is the sum of the distances between consecutive nodes encountered along the route in the simulated network. For the fictitous routing scheme, the distance traveled is simply the distance between the source and the destination node. The goal of this experiment is to quantify the cost, in terms of distance traveled, of maintaining only small routing tables in Pastry.

The number of nodes varies between 1,000 and 100,000, and again $b = 4$, $|L| = 16$, $|M| = 32$ and 100,000 objects are inserted and retrieved. Figure 5 shows the total (cumulative) distance traveled using Pastry during the 100,000 retrieval operations (labeled "Pastry"), compared to the distance traveled if every node had a complete routing table (labeled "Complete routing tables"). With the complete routing table, it is assumed that each node has an entry for every other node in the system. The distance with the complete routing table is the distance between the source node and the node storing the same replica that is reached when using Pastry (not necessarily the closest replica).

The results show that the Pastry routes are only approximately 40% longer than those achieved with complete routing tables. Considering that the routing tables in Pastry contain only approximately $\lceil log_{2^b} N \rceil \times (2^b - 1)$ entries, this result is quite good. For 100,000 nodes the Pastry routing tables contain approximately 75 entries, compared to 99,999 in the case of complete routing tables.

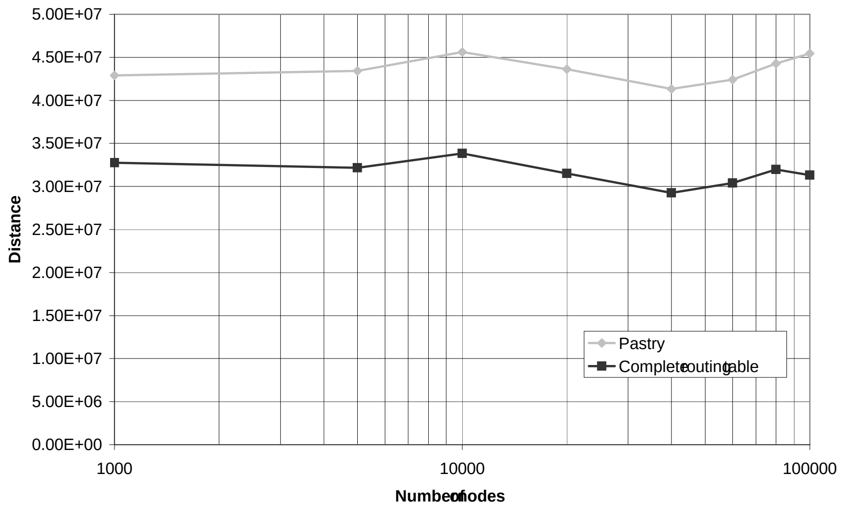

Fig. 5. Route distance versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$, and 100,000 objects.

We also determined the routing throughput, in messages per second, of a Pastry node. Our unoptimized Java implementation handled over 3,000 messages per second. This confirms that the routing procedure is very lightweight.

3.2 Maintaining the network

Figure 6 shows the quality of the routing tables, and how the extent of information exchange during a node join operation affects the quality of the resulting routing tables. In this experiment, 5,000 nodes join the Pastry network one by one. After all nodes joined, the routing tables were examined. The parameters are $b = 4, |L| = 16, |M| = 32$.

Three options were used to gather information when a node joins. "SL" is a hypothetical method where the joining node considers only the appropriate row from each node along the route from itself to the node with the closest existing nodeId (see Section 2.5). With "WT", the joining node fetches the entire state of each node along the path, but does not fetch state from the resulting representatives. This is equivalent to omitting the second stage. "WTF" is the actual method used in Pastry, where state is fetched from each node that appears in the tables after the first stage.

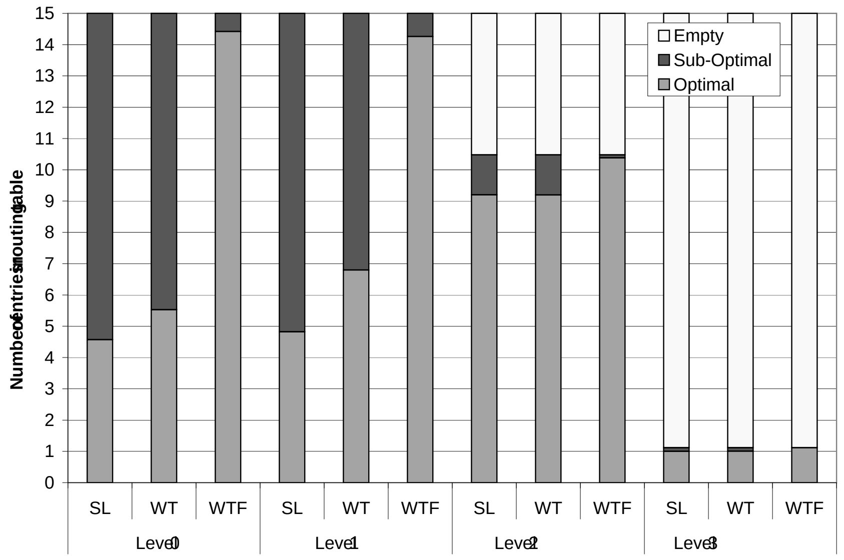

Fig. 6. Quality of routing tables, $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 nodes.

The results are shown in Figure 6. For levels 0 to 3, we show the quality of the routing table entries with each method. With 5,000 nodes and $b = 4$, levels 2 and 3 are not fully populated, which explains the missing entries shown. "Optimal" means that the best (closest in the network) representative appeared in the routing table, "sub-optimal" means that the representative was not the closest or was missing.

The results show that Pastry's method of node integration ("WTF") is highly effective in initializing the routing tables. On average, less than 1 entry per level of the routing able is not the best choice. Moreover, the comparison with "SL" and "WT" shows that less information exchange during the node join operation comes at a dramatic cost in routing table quality.

Node failures The next experiment explores Pastry's behaviour in the presence of node failures. In our experiment, 100,000 objects are inserted into a 5,000 node Pastry network with $b = 4$, $|L| = 16$, $|M| = 32$. Then, 10% (500) randomly selected nodes failed silently. After the failure, 2 lookups were performed for each of the objects (200,000 lookups total) from randomly selected nodes, while the node state repair facilities in Pastry were disabled. This allows us to measure the full impact of the failures on Pastry's routing performance. Next, the node state repair facilities were enabled, and another 200,000 lookups were performed from the same locations.

Figure 7 shows the average routing table quality across all nodes for levels 0–2, as measured before the failures, after the failures, and after the repair. Note that in this figure, missing entries are shown separately from sub-optimal entries. Also, recall that Pastry lazily repairs namespace set and routing tables entries when they are being used. As such, routing table entries that were not used during the 200,000 lookups are not discovered and therefore not repaired. To isolate the effectiveness of Pastry's repair procedure, we excluded table entries that were never used.

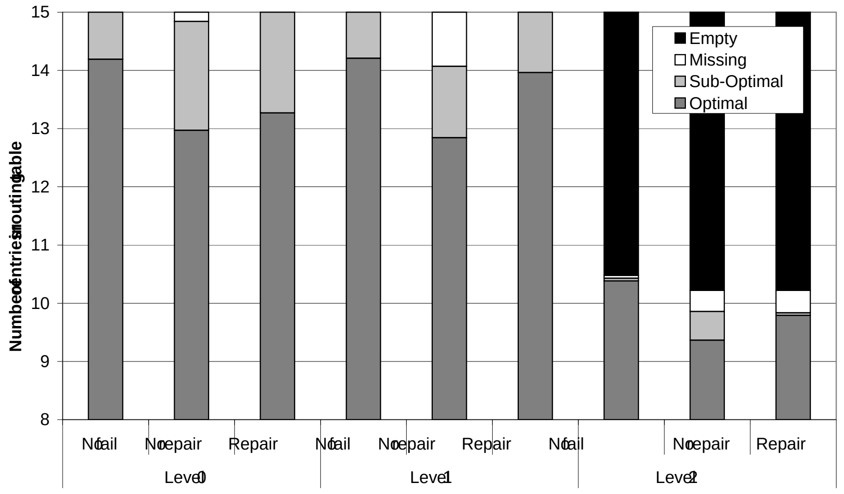

Fig. 7. Quality of routing tables before and after 500 node failures, $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 starting nodes.

The results show that Pastry recovers all missing table entries, and that the quality of the entries (fraction of optimal entries) approaches that before the failures. At level zero, the average number of best entries after the repair is approximately one below that prior to the failure. However, although this can't be seen in the figure, our results show that the actual distance between the suboptimal and the optimal representatives is very small. This is intuitive, since the average distance of level zero representatives is very small. Nevertheless, we are currently investigating a slight improvement to our repair procedure that we expect to improve this result.

Note that the increase in empty entries at levels 1 and 2 after the failures is due to the reduction in the total number of Pastry nodes, which increases the sparseness of the tables at the higher levels. Thus, this increase does not constitute a reduction in the quality of the tables.

Figure 8 shows the impact of failures and repairs on the route quality. The left bar shows the average number of hops before the failures; the middle bar shows the average number of hops after the failures, and before the tables were repaired. Finally, the right bar shows the average number of hops after the repair.

The data shows that without repairs, the stale routing table state causes as significant deterioration of route quality. After the repair, however, the average number of hops is only slightly higher than before the failures.

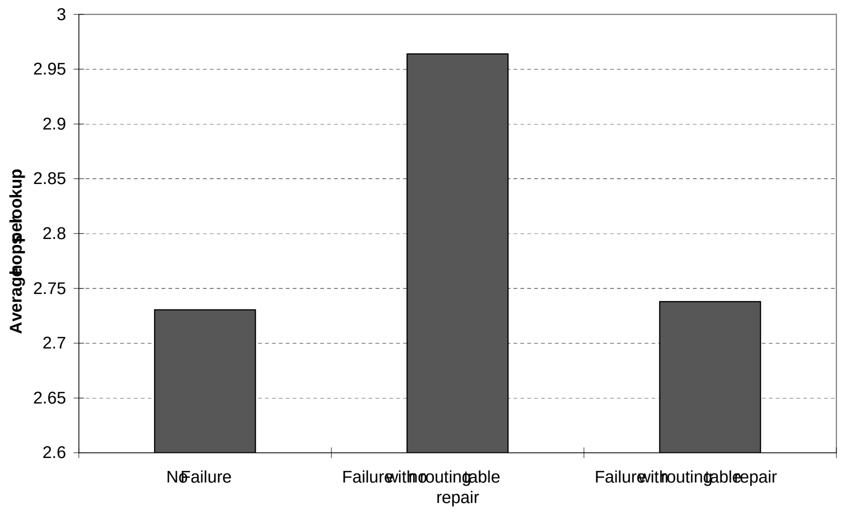

Fig. 8. Number of routing hops versus node failures, $b = 4$, $|L| = 16$, $|M| = 32$, 100,000 objects and 5,000 nodes with 500 failing.

We also measured the average cost, in messages, for repairing the tables after node failure. In our experiments, a total of 57 RPCs were needed on average per failed node to repair all relevant table entries.

3.3 Replica routing

The next experiment explores Pastry's ability to locate replicas near the client. In a Pastry network of 10,000 nodes with $b = 3$ and $|L| = 8$, 100,000 object are being inserted with 5 replicas each, and then looked up at randomly chosen nodes. Figure 9 shows the percentage of lookups that reached the closest replica (0 better replicas), the second closest replica (1 better replica), and so on. Results are shown for the three different protocols for initializing a new node's state, for the normal routing protocol as well as the heuristic mentioned in Section 2.4 and for an optimal version of the heuristic "Perfect estimation". The heuristic approach estimates the namespace coverage of other nodes namespace sets, using an estimate based on its own namespace sets coverage. Perfect estimation ensures that this estimate of a nodes namespace set coverage is correct for every node.

With the standard routing procedure and normal node joining protocol, Pastry is able to locate the closest replica 68% of the time, and one of the top two replicas 87% of the time. With the heuristic routing option, this figure increases to 76% and 92%, respectively. The lesser routing table quality resulting from the "SL" and "WT" methods for node joining have a strong negative effect on Pastry's ability to locate nearby replicas, as one would expect. Also, the results show that the heuristic approach is only approximately 2% worse than best possible results using perfect estimation.

The results show that Pastry is effective in locating a replica near the client in the vast majority of cases, and that the use of the heuristic improves the performance of Pastry. Furthermore, the heuristic performance is comparable to using perfect estimation.

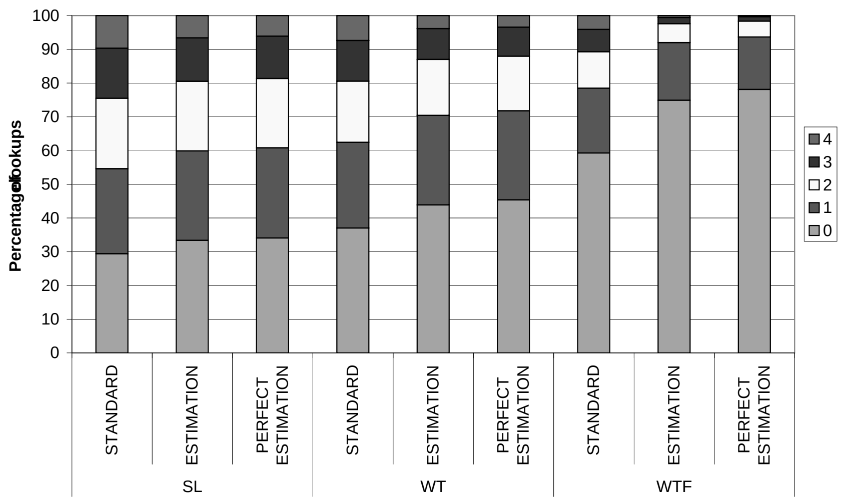

Fig. 9. Number of closer replicas to the client than the replica discovered. ($b = 3$, $|L| = 8$, $|M| = 16$, 10,000 nodes and 100,000 objects).

4 Related Work

There are currently many peer-to-peer systems under development that require highly scalable content location and request routing. Some of these systems are intended as file sharing facilities, such as Gnutella [3], Freenet [2], and Napster [1]. Whilst these systems have proved popular, their location and routing algorithms suffer from limitations. Napster uses a centralized document location discovery service, which limits its scalability. In Gnutella, the use of a broadcast based protocol limits the system's scalability and incurs a high bandwidth requirement. Furthermore, the routing algorithm does not guarantee to find an existing object. The same is also true for Freenet. These systems were designed for the large-scale sharing of mp3 files, and work under the assumption that there will be many replicas of a popular song, and therefore, the probability of finding it is high. This approach is not suitable for general peer-to-peer systems.

Pastry's routing scheme bears some similarity to the work by Plaxton et al. [6, 7]. The general approach of routing using prefix matching on the objId is used in both systems, which can be seen as a generalization of hypercube routing. However, there are important differences. In Plaxton et al.'s approach, there is a single node that holds information for a particular object, which makes the system susceptible to failure. In Pastry, on the other hand, replication is used for fault tolerance. Plaxton et al. also assume a static configuration, while Pastry assumes a dynamic system where nodes are free to join and leave at any time. Furthermore, in order to achieve good locality, Plaxton et al. assume knowledge of the location of all nodes in the system; Pastry merely assumes that a node can measure the distance from itself to another node.

There are a number of research projects focusing on peer-to-peer storage utilities, such as FarSite [4] and Oceanstore [5]. Farsite uses a distributed directory service to locate content. Unlike in Pastry, this location function is not integrated with the routing infrastructure.

Oceanstore uses a two phase approach to content location and routing. The first stage is probabilistic, using a generalization of Bloom filters. If that stage fails to find an object, then a location and routing scheme called Tapestry is used [8]. Tapestry is based on Plaxton et al. but extends that earlier work in several dimensions. Like Pastry, Tapestry replicates objects for fault resilience and availability and supports dynamic node addition and recovery from node failures. However, Pastry and Tapestry differ in the approach they take for replicating files and in the way they achieve locality.

There has been significant prior work on overlay networks. An overlay network consists of a collection of nodes placed strategically within an existing network infrastructure, and these nodes provide a network abstraction. As such, Pastry can be seen as an overlay network that provides an object discovery service. Another example of an overlay network is the Overcast system [9], which is one of several overlay networks aimed at providing reliable multicasting or content distribution. Overcast provides a single-source multicast stream distribution service.

There has been considerable work on routing in general, and of particular interest is the work on hypercube and mesh routing in parallel computers. Also, more recently the work on routing in ad hoc networks, for example GRID [10] and the routing algorithms used in PEN [11]. However, the challenges in developing routing algorithms for ad hoc networks differ, in as much as the main problem is device mobility. In Pastry, we assume that there is already a network infrastructure that is capable of routing messages between two nodes of the Pastry network, and the emphasis in on self-configuration and the integration of content location and routing.

In the interest of scalability, Pastry nodes only use local information, while traditional routing algorithms (like link-state and distance vector methods) globally propagate information about routes to each destination. This global information exchange limits the scalability of these routing algorithms, necessitating a hierarchical routing architecture like the one used in the Internet.

Several prior works consider issues in replicating Web content in the Internet, and selecting the nearest replica relative to a client HTTP query [12–14]. Pastry provides a more general infrastructure aimed at a variety of peer-to-peer applications.

Another related area is that of naming services, which are largely orthogonal to Pastry's content location and routing. Lampson's Global Naming System (GNS) [15] is an example of a scalable naming system that relies on a hierarchy of name servers that directly corresponds to the structure of the name space. Cheriton and Mann [16] describe another scalable naming service. Like GNS, their service is a pure naming service and relies on a hierarchy of name resolvers that reflects the structure of the name space.

Finally, attribute based and intentional naming systems [17, 18], as well as directory services [19, 20] resolve a set of attributes that describe the properties of an object to the address of an object instance that satisfies the given properties. Thus, these systems support far more powerful queries than Pastry. However, this power comes at the expense of scalability, performance and administrative overhead. Pastry supports the routing to a particular object identifier, rather than based on the properties of the object. Such systems could be potentially built upon Pastry.

5 Conclusion

We presented and evaluated Pastry, a generic, decentralized peer-to-peer content location and routing system for very large, self-configuring overlay networks of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably locates a copy of the requested content if one exists. Moreover, Pastry usually locates a copy of the requested content that is near the client. Pastry can be be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, and naming systems.

Pastry takes into account network locality when routing messages. In each routing step, Pastry chooses the nearest node that is closer in the namespace to the destination. Experimental results show that Pastry exhibits good network locality and that Pastry is usually able to locate the nearest copy of a replicated object. Additional experimental results with as many as 100,000 nodes show that Pastry scales well, that it is fully self-configuring and that it can gracefully adapt to node failures.

References

1. Napster. http://www.napster.com/.
2. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.
3. The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.
4. W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. SIGMETRICS'2000*, pages 34–43, 2000.

5. John Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, November 2000.
6. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 311–320, June 1997. Newport, Rhode Island, USA.
7. 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.
8. Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing, 2001. Submitted for publication.
9. John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI 2000*, October 2000.
10. Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, August 2000.
11. Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leask. Piconet - embedded mobile networking. *IEEE Personal Communications*, 4(5):8–15, October 1997.
12. Yair Amir, Alec Peterson, and David Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proceedings of the 12th International Symposium on Distributed Computing*, Andros, Greece, September 1998.
13. Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proceedings of the 4th Web Caching Workshop*, San Diego, CA, March 1999.
14. Jussi Kangasharju and Keith W. Ross. A replicated architecture for the domain name system. In *Proceedings of the IEEE Infocom 2000*, Tel Aviv, Israel, March 2000.
15. Butler Lampson. Designing a global name service. In *Proceedings of Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, August 1986.
16. David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.
17. Mic Bowman, Larry L. Peterson, and Andrey Yeatts. Univers: An attribute-based name server. *Software—Practice and Experience*, 20(4):403–424, April 1990.
18. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, Kiawah Island, SC, December 1999.
19. J. Reynolds. RFC 1309: Technical overview of directory services using the x.500 protocol, March 1992.
20. Mark A. Sheldon, Andrzej Duda, Ron Weiss, and David K. Gifford. Discover: A resource discovery system based on content routing. In *Proceedings of the 3rd International World Wide Web Conference*, 1995.