Pastry: Scalable, decentralized 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@cs.rice.edu

Abstract. This paper presents the design and evaluation of Pastry, a scalable, distributed object location and routing substrate 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 variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming.

Each node in the Pastry network has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a to scalar proximity metric like the number of IP routing hops.

Pastry is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes. Experimental results obtained with a prototype implementation on an emulated network of up to 100,000 nodes confirm Pastry's scalability and efficiency, its ability to self-organize and adapt to node failures, and its good network locality properties.

1 Introduction

Peer-to-peer Internet applications have recently been popularized through file sharing applications like Napster, Gnutella and FreeNet [1, 2, 8]. While much of the attention has been focused on the copyright issues raised by these particular applications, peer-to-peer systems have many interesting technical aspects like decentralized control, self-organization, adaptation and scalability. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric.

* Appears in Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). Heidelberg, Germany, November 2001.

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

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [1, 2, 5, 8, 10, 15]. One of the key problems in large-scale peer-to-peer applications is to provide efficient algorithms for object location and routing within the network. This paper presents Pastry, a generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliable. Moreover, Pastry has good route locality properties.

Pastry is intended as general substrate for the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems. Several application have been built on top of Pastry to date, including a global, persistent storage utility called PAST [11, 21] and a scalable publish/subscribe system called SCRIBE [22]. Other applications are under development.

Pastry provides the following capability. Each node in the Pastry network has a unique numeric identifier (nodeId). When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network. At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message.

Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Because nodeIds are randomly assigned, with high probability, the set of nodes with adjacent nodeId is diverse in geography, ownership, jurisdiction, etc. Applications can leverage this, as Pastry can route to one of k nodes that are numerically closest to the key. A heuristic ensures that among a set of nodes with the k closest nodeIds to the key, the message is likely to first reach a node "near" the node from which the message originates, in terms of the proximity metric.

Applications use these capabilities in different ways. PAST, for instance, uses a fileId, computed as the hash of the file's name and owner, as a Pastry key for a file. Replicas of the file are stored on the k Pastry nodes with nodeIds numerically closest to the fileId. A file can be looked up by sending a message via Pastry, using the fileId as the key. By definition, the lookup is guaranteed to reach a node that stores the file as long as one of the k nodes is live. Moreover, it follows that the message is likely to first reach a node near the client, among the k nodes; that node delivers the file and consumes the message. Pastry's notification mechanisms allow PAST to maintain replicas of a file on the k nodes closest to the key, despite node failure and node arrivals, and using only local coordination among nodes with adjacent nodeIds. Details on PAST's use of Pastry can be found in [11, 21].

As another sample application, in the SCRIBE publish/subscribe System, a list of subscribers is stored on the node with nodeId numerically closest to the topicId of a topic, where the topicId is a hash of the topic name. That node forms a rendez-vous point for publishers and subscribers. Subscribers send a message via Pastry using the topicId as the key; the registration is recorded at each node along the path. A publisher sends data to the rendez-vous point via Pastry, again using the topicId as the key. The rendez-vous point forwards the data along the multicast tree formed by the reverse paths from the rendez-vous point to all subscribers. Full details of Scribe's use of Pastry can be found in [22].

These and other applications currently under development were all built with little effort on top of the basic capability provided by Pastry. The rest of this paper is organized as follows. Section 2 presents the design of Pastry, including a description of the API. 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 interacts with local instances of one or more applications. 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.

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 nodeId space, which ranges from 0 to $2^{128} - 1$. The nodeId is assigned randomly when a node joins the system. It is assumed that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space. For instance, nodeIds could be generated by computing a cryptographic hash of the node's public key or its IP address. As a result of this random assignment of nodeIds, with high probability, nodes with adjacent nodeIds are diverse in geography, ownership, jurisdiction, network attachment, etc.

Assuming a network consisting of N nodes, Pastry can route to the numerically closest node to a given key in less than $\lceil log_{2^b} N \rceil$ steps under normal operation (b is a configuration parameter with typical value 4). Despite concurrent node failures, eventual delivery is guaranteed unless $\lfloor |L|/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously ($|L|$ is a configuration parameter with a typical value of 16 or 32). In the following, we present the Pastry scheme.

For the purpose of routing, nodeIds and keys are thought of as a sequence of digits with base 2^b. Pastry routes messages to the node whose nodeId is numerically closest to the given key. This is accomplished as follows. In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. To support this routing procedure, each node maintains some routing state, which we describe next.

2.1 Pastry node state

Each Pastry node maintains a *routing table*, a *neighborhood set* and a *leaf set*. We begin with a description of the routing table. A node's routing table, R, is organized into

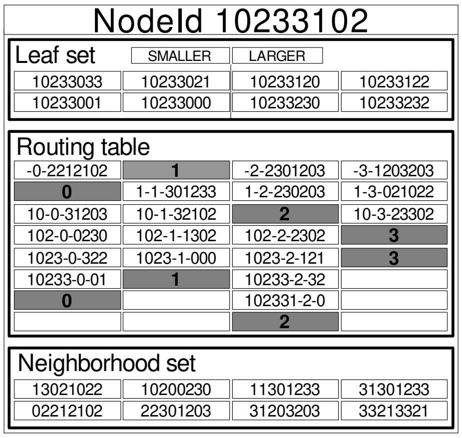

Fig. 1. State of a hypothetical Pastry node with nodeId 10233102, $b = 2$, and $l = 8$. All numbers are in base 4. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102 - next digit - rest of nodeId*. The associated IP addresses are not shown.

$\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries at row n of the routing table each refer to a node whose nodeId shares the present node's nodeId in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n + 1$th digit in the present node's id.

Each entry in the routing table contains the IP address of one of potentially many nodes whose nodeId have the appropriate prefix; in practice, a node is chosen that is close to the present node, according to the proximity metric. We will show in Section 2.5 that this choice provides good locality properties. If no node is known with a suitable nodeId, then the routing table entry is left empty. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, on average, only $\lceil log_{2^b} N \rceil$ rows are populated in the routing table.

The choice of b involves a trade-off between the size of the populated portion of the routing table (approximately $\lceil log_{2^b} N \rceil \times (2^b - 1)$ entries) and the maximum number of hops required to route between any pair of nodes ($\lceil log_{2^b} N \rceil$). With a value of $b = 4$ and 10^6 nodes, a routing table contains on average 75 entries and the expected number of routing hops is 5, whilst with 10^9 nodes, the routing table contains on average 105 entries, and the expected number of routing hops in 7.

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; it is useful in maintaining locality properties, as discussed in Section 2.5. The leaf set L is the set of nodes with the $|L|/2$ numerically closest larger nodeIds, and the $|L|/2$ nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. The leaf set is used during the message routing, as described below. Typical values for $|L|$ and $|M|$ are 2^b or 2×2^b.

How the various tables of a Pastry node are initialized and maintained is the subject of Section 2.4. 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 $b = 2$.

2.2 Routing

The Pastry routing procedure is shown in pseudo code form in Table 1. The procedure is executed whenever a message with key D arrives at a node with nodeId A. We begin by defining some notation.
R_l^i: the entry in the routing table R at column i, $0 \le i < 2^b$ and row l, $0 \le l < \lfloor 128/b \rfloor$.
L_i: the i-th closest nodeId in the leaf set L, $-\lfloor |L|/2 \rfloor \le i \le \lfloor |L|/2 \rfloor$, where negative/positive indices indicate nodeIds smaller/larger than the present nodeId, respectively.
D_l: the value of the l's digit in the key D.
$shl(A, B)$: the length of the prefix shared among A and B, in digits.

(1) if ($L_{-\lfloor |L|/2 \rfloor} \le D \le L_{\lfloor |L|/2 \rfloor}$) {
(2)   // D is within range of our leaf set
(3)   forward to L_i, s.th. $|D - L_i|$ is minimal;
(4) } else {
(5)   // use the routing table
(6)   Let $l = shl(D, A)$;
(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) $shl(T, D) \ge l$,
(14) $|T - D| < |A - D|$
(15)   }
(16) }

Table 1. Pseudo code for Pastry core routing algorithm.

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

If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit (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 key at least as long as the local node, and is numerically closer to the key than the present node's id.

Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless $\lfloor |L|/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be live.

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

Routing performance It can be shown that the expected number of routing steps is $\lceil log_{2^b} N \rceil$ steps, assuming accurate routing tables and no recent node failures. 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 whose ids have a longer prefix match with the key 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 key is within range of the leaf set (lines 2–3), then the destination node is at most one hop away.

The third case arises when the key is not covered by the leaf set (i.e., it is still more than one hop away from the destination), but there is no routing table entry. Assuming accurate routing tables and no recent node failures, this means that a node with the appropriate prefix does not exist (lines 11–14). The likelihood of this case, given the uniform distribution of nodeIds, depends on $|L|$. Analysis shows that with $|L| = 2^b$ and $|L| = 2 \times 2^b$, the probability that this case arises during a given message transmission is less than .02 and 0.006, respectively. When it happens, no more than one additional routing step results with high probability.

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. This is a loose upper bound; in practice, routing performance degrades gradually with the number of recent node failures, as we will show experimentally in Section 3.1. Eventual message delivery is guaranteed unless $\lfloor |L|/2 \rfloor$ nodes with consecutive nodeIds fail simultaneously. Due to the expected diversity of nodes with adjacent nodeIds, and with a reasonable choice for $|L|$ (e.g. 2^b), the probability of such a failure can be made very low.

2.3 Pastry API

Next, we briefly outline Pastry's application programming interface (API). The presented API is slightly simplified for clarity. Pastry exports the following operations:

nodeId = pastryInit(Credentials, Application) causes the local node to join an existing Pastry network (or start a new one), initialize all relevant state, and return the local node's nodeId. The application-specific credentials contain information needed to authenticate the local node. The application argument is a handle to the application object that provides the Pastry node with the procedures to invoke when certain events happen, e.g., a message arrival.

route(msg,key) causes Pastry to route the given message to the node with nodeId numerically closest to the key, among all live Pastry nodes.

Applications layered on top of Pastry must export the following operations:

deliver(msg,key) called by Pastry when a message is received and the local node's nodeId is numerically closest to key, among all live nodes.

forward(msg,key,nextId) called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL terminates the message at the local node.

newLeafs(leafSet) called by Pastry whenever there is a change in the local node's leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

Several applications have been built on top of Pastry using this simple API, including PAST [11, 21] and SCRIBE [22], and several applications are under development.

2.4 Self-organization and adaptation

In this section, we describe Pastry's protocols for handling the arrival and departure of nodes in the Pastry network. We begin with the arrival of a new node that joins the system. Aspects of this process pertaining to the locality properties of the routing tables are discussed in Section 2.5.

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

Let us assume the new node's nodeId is X. (The assignment of nodeIds is application-specific; typically it is computed as the SHA-1 hash of its IP address or its public key). Node X then asks A to route a special "join" message with the key equal to X. Like any message, Pastry routes the join message 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 tables to X. The new node X inspects this information, may request state from additional nodes, and then initializes its own state tables, using a procedure describe below. Finally, X informs any nodes that need to be aware of its arrival. This procedure ensures that X initializes its state with appropriate values, and that the state in all other affected nodes is updated.

Since node A is assumed to be in proximity to the new node X, A's neighborhood set to initialize X's neighborhood set. Moreover, Z has the closest existing nodeId to X, thus its leaf set is the basis for X's leaf set. Next, we consider the routing table, starting at row zero. We consider the most general case, where the nodeIds of A and X share no common prefix. Let A_i denote node A's row of the routing table at level i. Note that the entries in row zero of the routing table are independent of a node's nodeId. Thus, A_0 contains appropriate 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 entries in B_1 and

X_1 share the same prefix, because X and B have the same first digit in their nodeId. Similarly, X obtains appropriate entries for X_2 from node C, the next node encountered along the route from A to Z, and so on.

Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing table. Those nodes in turn update their own state based on the information received. One can show that at this stage, the new node X is able to route and receive messages, and participate in the Pastry network. The total cost for a node join, in terms of the number of messages exchanged, is $O(log_{2^b} N)$. The constant is about 3×2^b.

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. A Pastry node is considered failed when its immediate neighbors in the nodeId space can no longer communicate with the node.

To replace a failed node in the leaf set, its neighbor in the nodeId space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf table. For instance, if L_i failed for $\lfloor |L|/2 \rfloor < i < 0$, it requests the leaf set from $L_{-\lfloor |L|/2 \rfloor}$. Let the received leaf set be L'. This set partly overlaps the present node's leaf 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 leaf set unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeIds have failed simultaneously. Due to the diversity of nodes with adjacent nodeIds, such a failure is very unlikely even for modest values of $|L|$.

The failure of a node that appears in the routing table of another node is detected when that node attempts to contact the failed node and there is no response. As explained in Section 2.2, this event does not normally delay the routing of a message, since the message can be forwarded to another node. However, a replacement entry must be found to preserve the integrity of the routing table.

To repair a failed routing table entry R_l^d, a node contacts first the node referred to by another entry $R_l^i, i \neq d$ of the same row, and asks for that node's entry for R_l^d. In the event that none of the entries in row l have a pointer to a live node with the appropriate prefix, the node next contacts an entry $R_{l+1}^i, i \neq d$, thereby casting a wider net. This procedure is highly likely to eventually find an appropriate node if one exists.

The neighborhood set is not normally used in the routing of messages, yet it is important to keep it current, because the set plays an important role in exchanging information about nearby nodes. For this purpose, a node attempts to contact each member of the neighborhood set periodically to see if it is still alive. 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.

2.5 Locality

In the previous sections, we discussed Pastry's basic routing properties and discussed its performance in terms of the expected number of routing hops and the number of messages exchanged as part of a node join operation. This section focuses on another aspect of Pastry's routing performance, namely its properties with respect to locality. We will show that the route chosen for a message is likely to be "good" with respect to the proximity metric.

Pastry's notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance. It is assumed 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 is expected to implements this function depending on its choice of a proximity metric, using network services like traceroute or Internet subnet maps, and appropriate caching and approximation techniques to minimize overhead.

Throughout this discussion, we assume that the proximity space defined by the chosen proximity metric is Euclidean; that is, the triangulation inequality holds for distances among Pastry nodes. This assumption does not hold in practice for some proximity metrics, such as the number of IP routing hops in the Internet. If the triangulation inequality does not hold, Pastry's basic routing is not affected; however, the locality properties of Pastry routes may suffer. Quantifying the impact of such deviations is the subject of ongoing work.

We begin by describing how the previously described procedure for node arrival is augmented with a heuristic that ensures that routing table entries are chosen to provide good locality properties.

Locality in the routing table In Section 2.4, we described how a newly joining node initializes its routing table. Recall that a newly joining node X asks an existing node A to route a join message using X as the key. The message follows a paths through nodes A, B, etc., and eventually reaches node Z, which is the live node with the numerically closest nodeId to X. Node X initialized its routing table by obtaining the i-th row of its routing table from the i-th node encountered along the route from A to Z.

The property we wish to maintain is that all routing table entries refer to a node that is near the present node, according to the proximity metric, among all live nodes with a prefix appropriate for the entry. Let us assume that this property holds prior to node X's joining the system, and show how we can maintains the property as node X joins.

First, we require that node A is near X, according to the proximity metric. Since the entries in row zero of A's routing table are close to A, A is close to X, and we assume that the triangulation inequality holds in the proximity space, it follows that the entries are relatively near A. Therefore, the desired property is preserved. Likewise, obtaining X's neighborhood set from A is appropriate.

Let us next consider row one of X's routing table, which is obtained from node B. The entries in this row are near B, however, it is not clear how close B is to X. Intuitively, it would appear that for X to take row one of its routing table from node B does not preserve the desired property, since the entries are close to B, but not necessarily to X. In reality, the entries tend to be reasonably close to X. Recall that the entries in each successive row are chosen from an exponentially decreasing set size. Therefore, the expected distance from B to its row one entries (B_1) is much larger than the expected distance traveled from node A to B. As a result, B_1 is a reasonable choice for X_1. This same argument applies for each successive level and routing step, as depicted in Figure 2.

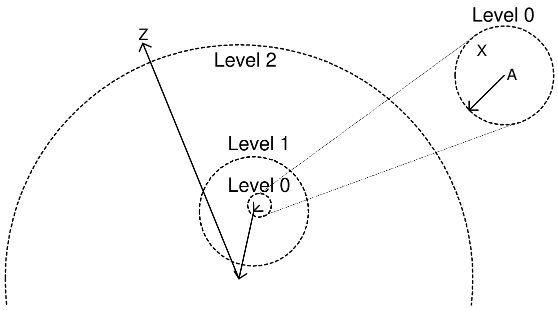

Fig. 2. Routing step distance versus distance of the representatives at each level (based on experimental data). The circles around the n-th node along the route from A to Z indicate the average distance of the node's representatives at level n. Note that X lies within each circle.

After X has initialized its state in this fashion, its routing table and neighborhood set approximate the desired locality property. However, the quality of this approximation 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 entries found in those nodes' routing tables and neighborhood sets, respectively, and updates its own state with any closer nodes it finds. The neighborhood set contributes valuable information in this process, because it maintains and propagates information about nearby nodes regardless of their nodeId prefix.

Intuitively, a look at Figure 2 illuminates why incorporating the state of nodes mentioned in the routing and neighborhood tables from stage one provides good representatives for X. The circles show the average distance of the entry from each node along the route, corresponding to the rows in the routing table. Observe that X lies within each circle, albeit off-center. In the second stage, X obtains the state from the entries discovered in stage one, which are located at an average distance equal to the perimeter of each respective circle. These states must include entries that are appropriate for X, but were not discovered by X in stage one, due to its off-center location.

Experimental results in Section 3.2 show that this procedure maintains the locality property in the routing table and neighborhood sets with high fidelity. Next, we discuss how the locality in Pastry's routing tables affects Pastry routes.

Route locality The entries in the routing table of each Pastry node are chosen to be close to the present node, according to the proximity metric, among all nodes with the desired nodeId prefix. As a result, in each routing step, a message is forwarded to a relatively close node with a nodeId that shares a longer common prefix or is numerically closer to the key than the local node. That is, each step moves the message closer to the destination in the nodeId space, while traveling the least possible distance in the proximity space.

Since only local information is used, Pastry minimizes the distance of the next routing step with no sense of global direction. This procedure clearly does not guarantee that the shortest path from source to destination is chosen; however, it does give rise to relatively 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 an entry in the routing table in row l is chosen from a set of nodes of size $N/2^{bl}$. That is, the entries in successive rows 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 entry in each successive row 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 3 illustrates this effect.

Locating the nearest among k nodes Some peer-to-peer application we have built using Pastry replicate information on the k Pastry nodes with the numerically closest nodeIds to a key in the Pastry nodeId space. PAST, for instance, replicates files in this way to ensure high availability despite node failures. Pastry naturally routes a message with the given key to the live node with the numerically closest nodeId, thus ensuring that the message reaches one of the k nodes as long as at least one of them is live.

Moreover, Pastry's locality properties make it likely that, along the route from a client to the numerically closest node, the message first reaches a node near the client, in terms of the proximity metric, among the k numerically closest nodes. This is useful in applications such as PAST, because retrieving a file from a nearby node minimizes client latency and network load. Moreover, 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 node that is located relatively near the client.

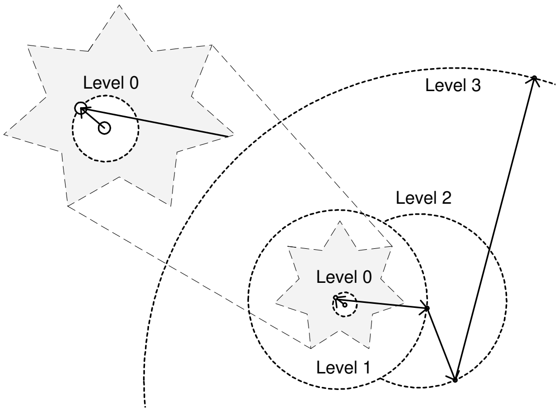

Fig. 3. Sample trajectory of a typical message in the Pastry network, based on experimental data. The message cannot re-enter the circles representing the distance of each of its routing steps away from intermediate nodes. Although the message may partly "turn back" during its initial steps, the exponentially increasing distances traveled in each step cause it to move toward its destination quickly.

Recall that Pastry routes messages towards the node with the nodeId closest to the key, while attempting to travel the smallest possible distance in each step. Therefore, among the k numerically closest nodes to a key, a message tends to first reach a node near the client. Of course, this process only approximates routing to the nearest node. Firstly, as discussed above, Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no sense of global direction. Secondly, since Pastry routes primarily based on nodeId prefixes, it may miss nearby nodes with a different prefix than the key. In the worst case, $k/2 - 1$ of the replicas are stored on nodes whose nodeIds differ from the key in their domain at level zero. As a result, Pastry will first route towards the nearest among the $k/2 + 1$ remaining nodes.

Pastry uses a heuristic to overcome the prefix mismatch issue described above. The heuristic is based on estimating the density of nodeIds in the nodeId space using local information. Based on this estimation, the heuristic detects when a message approaches the set of k numerically closest nodes, and then switches to numerically nearest address based routing to locate the nearest replica. Results presented in Section 3.3 show that Pastry is able to locate the nearest node in over 75%, and one of the two nearest nodes in over 91% of all queries.

2.6 Arbitrary node failures and network partitions

Throughout this paper, it is assumed that Pastry nodes fail silently. Here, we briefly discuss how a Pastry network could deal with arbitrary nodes failures, where a failed node continues to be responsive, but behaves incorrectly or even maliciously. The Pastry routing scheme as described so far is deterministic. Thus, it is vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they normally take the same route.

In applications where arbitrary node failures must be tolerated, the routing can be randomized. Recall that in order to avoid routing loops, a message must always be forwarded to a node that shares a longer prefix with the destination, or shares the same prefix length as the current node but is numerically closer in the nodeId space than the current node. However, the choice among multiple nodes that satisfy this criterion can be made randomly. In practice, the probability distribution should be biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node. Furthermore, the protocols for node join and node failure can be extended to tolerate misbehaving nodes. The details are beyond the scope of this paper.

Another challenge are IP routing anomalies in the Internet that cause IP hosts to be unreachable from certain IP hosts but not others. The Pastry routing is tolerant of such anomalies; Pastry nodes are considered live and remain reachable in the overlay network as long as they are able to communication with their immediate neighbors in the nodeId space. However, Pastry's self-organization protocol may cause the creation of multiple, isolated Pastry overlay networks during periods of IP routing failures. Because Pastry relies almost exclusively on information exchange within the overlay network to self-organize, such isolated overlays may persist after full IP connectivity resumes.

One solution to this problem involves the use of IP multicast. Pastry nodes can periodically perform an expanding ring multicast search for other Pastry nodes in their vicinity. If isolated Pastry overlays exists, they will be discovered eventually, and reintegrated. To minimize the cost, this procedure can be performed randomly and infrequently by Pastry nodes, only within a limited range of IP routing hops from the node, and only if no search was performed by another nearby Pastry node recently. As an added benefit, the results of this search can also be used to improve the quality of the routing tables.

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 emulation environment, permitting experiments with up to 100,000 Pastry nodes.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 6GBytes 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.

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 emulated network environment maintains distance information between the Pastry nodes. Each Pastry node is assigned a location in a plane; coordinates in the plane are randomly assigned in the range $[0, 1000]$. Nodes in the Internet are not uniformly distributed in a Euclidean space; instead, there is a strong clustering of nodes and the triangulation inequality doesn't always hold. We are currently performing emulations based on a more realistic network topology model taken from [26]. Early results indicate that overall, Pastry's locality related routing properties are not significantly affected by this change.

A number of Pastry properties are evaluated experimentally. The first set of results demonstrates the basic performance of Pastry routing. The routing tables created within the Pastry nodes are evaluated in Section 3.2. In Section 3.3 we evaluate Pastry's ability to route to the nearest among the k numerically closest nodes to a key. Finally, in 3.4 the properties of Pastry under node failures are considered.

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 $b = 4$, $|L| = 16$, $|M| = 32$. In each of 200,000 trials, two Pastry nodes are selected at random and a message is routed between the pair using Pastry.

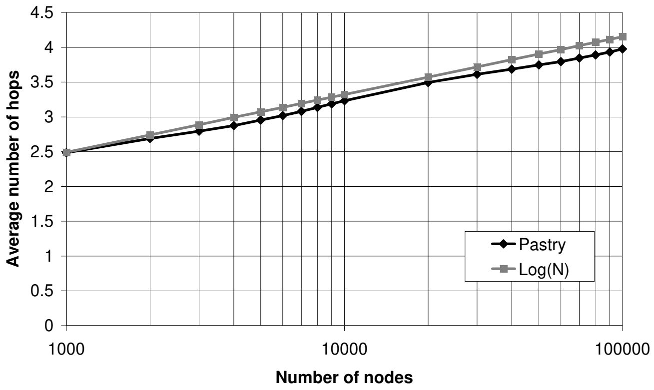

Fig. 4. Average number of routing hops versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$ and 200,000 lookups.

Figure 4 show the average number of routing hops taken, as a function of the network size. "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 predicted.

Figure 5 shows the distribution of the number of routing hops taken, for a network size of 100,000 nodes, in the same experiment. The results show that the maximum route length is ($\lceil log_{2^b} N \rceil$ ($\lceil log_{2^b} 100,000 \rceil = 5$), as expected.

The second experiment evaluated the locality properties of Pastry routes. It compares the relative distance a message travels using Pastry, according to the proximity

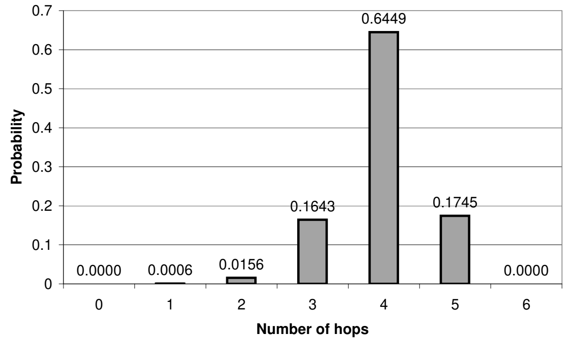

Fig. 5. Probability versus number of routing hops, $b = 4$, $|L| = 16$, $|M| = 32$, $N = 100,000$ and 200,000 lookups.

metric, with that of a fictitious routing scheme that maintains complete routing tables. The distance traveled is the sum of the distances between consecutive nodes encountered along the route in the emulated network. For the fictitious routing scheme, the distance traveled is simply the distance between the source and the destination node. The results are normalized to the distance traveled in the fictitious routing scheme. The goal of this experiment is to quantify the cost, in terms of distance traveled in the proximity space, of maintaining only small routing tables in Pastry.

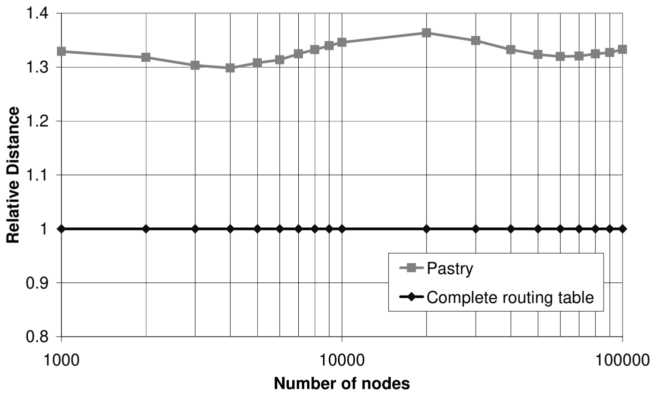

Fig. 6. Route distance versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$, and 200,000 lookups.

The number of nodes varies between 1,000 and 100,000, $b = 4$, $|L| = 16$, $|M| = 32$. 200,000 pairs of Pastry nodes are selected and a message is routed between each pair. Figure 6 shows the results for Pastry and the fictitious scheme (labeled "Complete routing tables"). The results show that the Pastry routes are only approximately 30% to 40% longer. 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.

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 indicates that the routing procedure is very lightweight.

3.2 Maintaining the network

Figure 7 shows the quality of the routing tables with respect to the locality property, and how the extent of information exchange during a node join operation affects the quality of the resulting routing tables vis-à-vis locality. 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 2.4). With “WT”, the joining node fetches the entire state of each node along the path, but does not fetch state from the resulting entries. 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.

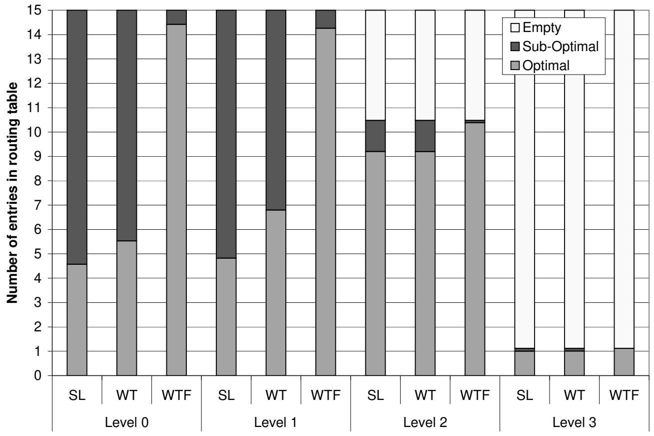

Fig. 7. Quality of routing tables (locality), $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 nodes.

The results are shown in Figure 7. 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 (i.e., closest according to the proximity metric) node appeared in a routing table entry, “sub-optimal” means that an entry 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 with good locality. 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 with respect to locality.

3.3 Replica routing

The next experiment examines Pastry's ability to route to one of the k closest nodes to a key, where $k = 5$. In particular, the experiment explores Pastry's ability to locate one of the k nodes near the client. In a Pastry network of 10,000 nodes with $b = 3$ and $|L| = 8$, 100,000 times a Pastry node and a key are chosen randomly, and a message is routed using Pastry from the node using the key. The first of the k numerically closest nodes to the key that is reached along the route is recorded.

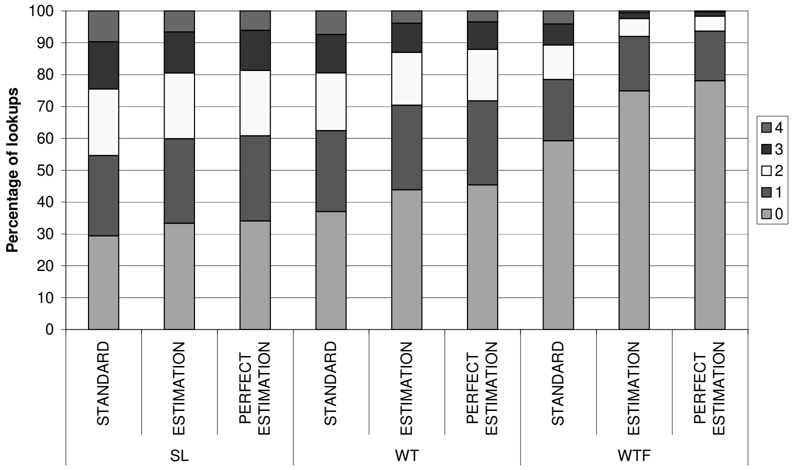

Fig. 8. Number of nodes closer to the client than the node discovered. ($b = 3$, $|L| = 8$, $|M| = 16$, 10,000 nodes and 100,000 message routes).

Figure 8 shows the percentage of lookups that reached the closest node, according to the proximity metric (0 closer nodes), the second closest node (1 closer node), and so forth. Results are shown for the three different protocols for initializing a new node's state, with ("Estimation") and without ("Standard") the heuristic mentioned in Section 2.5, and for an idealized, optimal version of the heuristic ("Perfect estimation"). Recall that the heuristic estimates the nodeId space coverage of other nodes' leaf sets, using an estimate based on its own leaf sets coverage. The "Perfect estimation" ensures that this estimate of a node's leaf set coverage is correct for every node.

Without the heuristic and the standard node joining protocol (WTF), Pastry is able to locate the closest node 68% of the time, and one of the top two nodes 87% of the time. With the heuristic routing option, this figures increase 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 nodes, as one would expect. Also, the heuristic approach is only approximately 2% worse than the best possible results using perfect estimation.

The results show that Pastry is effective in locating a node near the client in the vast majority of cases, and that the use of the heuristic is effective.

3.4 Node failures

The next experiment explores Pastry's behavior in the presence of node failures. A 5,000 node Pastry network is used with $b = 4$, $|L| = 16$, $|M| = 32$, $k = 5$. Then, 10% (500) randomly selected nodes fail silently. After these failures, a key is chosen at random, and two Pastry nodes are randomly selected. A message is routed from these two nodes to the key, and this is repeated 100,000 times (200,000 lookups total). Initially, the node state repair facilities in Pastry were disabled, which 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 Pastry nodes to the same key.

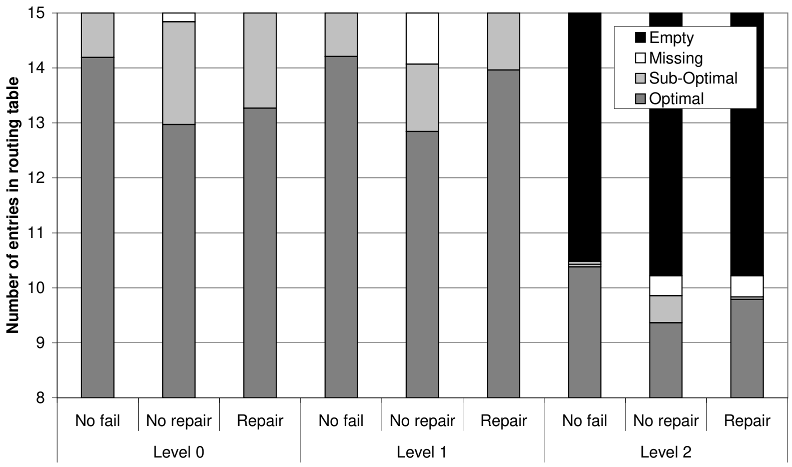

Fig. 9. Quality of routing tables before and after 500 node failures, $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 starting nodes.

Figure 9 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 routing tables entries when they are being used. As a result, 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.

The results show that Pastry recovers all missing table entries, and that the quality of the entries with respect to locality (fraction of optimal entries) approaches that before the failures. At row 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 entries is very small. This is intuitive, since the average distance of row zero entries is very small. 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 upper rows. Thus, this increase does not constitute a reduction in the quality of the tables.

Figure 10 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.

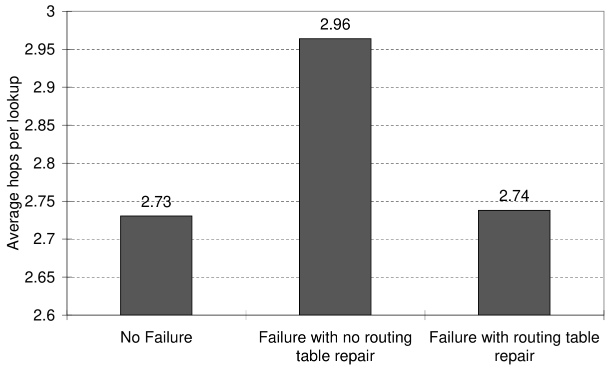

Fig. 10. Number of routing hops versus node failures, $b = 4$, $|L| = 16$, $|M| = 32$, 200,000 lookups and 5,000 nodes with 500 failing.

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

4 Related Work

There are currently several peer-to-peer systems in use, and many more are under development. Among the most prominent are file sharing facilities, such as Gnutella [2] and Freenet [8]. The Napster [1] music exchange service provided much of the original motivation for peer-to-peer systems, but it is not a pure peer-to-peer system because its database is centralized. All three systems are primarily intended for the large-scale sharing of data files; reliable content location is not guaranteed or necessary in this environment. In Gnutella, the use of a broadcast based protocol limits the system's scalability and incurs a high bandwidth requirement. Both Gnutella and Freenet are not guaranteed to find an existing object.

Pastry, along with Tapestry [27], Chord [24] and CAN [19], represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella.

Pastry and Tapestry bear some similarity to the work by Plaxton et al [18] and to routing in the landmark hierarchy [25]. The approach of routing based on address prefixes, which can be viewed as a generalization of hypercube routing, is common to all these schemes. However, neither Plaxton nor the landmark approach are fully self-organizing. Pastry and Tapestry differ in their approach to achieving network locality and to supporting replication, and Pastry appears to be less complex.

The Chord protocol is closely related to both Pastry and Tapestry, but instead of routing towards nodes that share successively longer address prefixes with the destination, Chord forwards messages based on numerical difference with the destination address. Unlike Pastry and Tapestry, Chord makes no explicit effort to achieve good network locality. CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. Unlike Pastry, the routing table does not grow with the network size, but the number of routing hops grows faster than $logN$.

Existing applications built on top of Pastry include PAST [11, 21] and SCRIBE [22]. Other peer-to-peer applications that were built on top of a generic routing and location substrate like Pastry are OceanStore [15] (Tapestry) and CFS [9] (Chord). FarSite [5] uses a conventional distributed directory service, but could potentially be built on top of a system like Pastry. Pastry can be seen as an overlay network that provides a self-organizing routing and location service. Another example of an overlay network is the Overcast system [12], which provides reliable single-source multicast streams.

There has been considerable work on routing in general, on hypercube and mesh routing in parallel computers, and more recently on routing in ad hoc networks, for example GRID [17]. In Pastry, we assume an existing infrastructure at the network layer, and the emphasis in on self-organization 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 [4, 13, 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) [16] 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 [7] describe another scalable naming service.

Finally, attribute based and intentional naming systems [6, 3], as well as directory services [20, 23] 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. Such systems could be potentially built upon Pastry.

5 Conclusion

This paper presents and evaluates Pastry, a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably routes a message to the live node with a nodeId numerically closest to a key. Pastry can be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems.

Pastry routes to any node in the overlay network in O(log N) steps in the absence of recent node failures, and it maintains routing tables with only O(log N) entries. Moreover, Pastry takes into account locality when routing messages. Results with as many as 100,000 nodes in an emulated network confirm that Pastry is efficient and scales well, that it is self-organizing and can gracefully adapt to node failures, and that it has good locality properties.

Acknowledgments

We thank Miguel Castro and the anonymous reviewers for their useful comments and feedback. Peter Druschel thanks Microsoft Research, Cambridge, UK, and the Massachusetts Institute of Technology for their support during his visits in Fall 2000 and Spring 2001, respectively, and Compaq for donating equipment used in this work.

References

1. Napster. http://www.napster.com/.
2. The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.
3. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. SOSP'99*, Kiawah Island, SC, Dec. 1999.
4. Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proc. 12th Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.
5. 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*, Santa Clara, CA, 2000.
6. M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software — Practice and Experience*, 20(4):403–424, Apr. 1990.

7. D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Trans. Comput. Syst.*, 7(2):147–183, May 1989.
8. I. Clarke, O. Sandberg, B. Wiley, and T. 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.
9. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.
10. R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
11. P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, Schloss Elmau, Germany, May 2001.
12. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI 2000*, San Diego, CA, 2000.
13. J. Kangasharju, J. W. Roberts, and K. W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proc. 4th Web Caching Workshop*, San Diego, CA, Mar. 1999.
14. J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proc. IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.
15. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.
16. B. Lampson. Designing a global name service. In *Proc. Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, Minaki, Canada, Aug. 1986.
17. J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, Boston, MA, 2000.
18. 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.
19. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.
20. J. Reynolds. RFC 1309: Technical overview of directory services using the X.500 protocol, Mar. 1992.
21. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.
22. A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. Submitted for publication. June 2001. http://www.research.microsoft.com/ antr/SCRIBE/.
23. M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proc. 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.
24. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.
25. P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *SIGCOMM'88*, Stanford, CA, 1988.
26. E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, San Francisco, CA, 1996.
27. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.