Pastel: Bridging the Gap Between Structured and Large-State Overlays

Nuno Cruces, Rodrigo Rodrigues, Paulo Ferreira
INESC-ID, Lisbon, Portugal

Abstract

Peer-to-peer overlays envision a single overlay substrate that can be used (possibly simultaneously) by many applications, but current overlays either target fast, few-hop lookups for contacting directly the responsible nodes, or slower multi-hop lookups that can be used by applications that exploit the overlay topology (like multicast or anycast). In this paper we present Pastel, an extension to Pastry that bridges the gap between the two types of overlays. Pastel maintains both Pastry routing tables and a full information table, and we show how we can exploit synergies between the maintenance of the two. We also propose a novel API that is richer than the one offered by existing overlays, to give applications control over the type of lookups (structured, multi-hop routing, or attempt direct contact).

We implemented Pastel in a discrete-event packet level simulator and our results show that Pastel has lookups that are usually more efficient than Pastry's. Furthermore, the bandwidth required by Pastel is modest, even for a system with thousands of nodes.

1 Introduction

Peer-to-peer overlays (like Chord [24], Pastry [21], Tapestry [11], or CAN [18]) form a decentralized, self-organizing substrate that can be used by a myriad of different applications with distinct requirements and ways of using the overlay. In many cases, the designers of such overlays have expressed the vision of deploying a single peer-to-peer overlay with many applications running on top of it (e.g., [12]).

We can categorize the applications that have been proposed to run on top of peer-to-peer overlays in two broad groups.

The first group consists of *direct contact* applications that typically use a narrow get/put interface offered by a layer running on top of a routing overlay that implements a distributed hash table interface (DHT) [7,22]. These applications (or, in some cases, the DHTs that underlie them) only use the routing overlay to locate the node or set of nodes that are responsible for storing a particular data item. Once the lookup primitive returns the node or set of nodes that are responsible for the information, these nodes are contacted directly to store or retrieve the data. Examples of such applications include file systems [15], or databases for citation indices [25].

The second group consists of *routing* applications, which are applications that actively use topology formed by the routing overlay. For example, a multicast application will form trees by taking the union of the lookup paths to a common identifier [3]. The same principle is used by anycast applications [4].

In previous work, authors have presented proposals for reducing the lookup latency of peer-to-peer overlays by increasing the amount of routing state maintained by each node [10, 13, 17, 19]. Such overlays achieve faster lookups because increasing the knowledge of each node about other members of the overlay will lead to a shorter lookup path, or ultimately preclude routing (i.e., a full information or "one-hop" lookup). However, such overlays cannot be used by routing applications, where multi-hop lookup paths are required.

This divide between overlays that support multi-hop lookups and overlays that keep a large routing state conflicts with the vision of a single overlay that can support multiple applications (simultaneously or not). This is because routing applications cannot be deployed in overlays with large routing state (since they lack both the interface to contact

nodes along a lookup path and the topology formed by structured overlay), and direct contact applications pay the penalty of slow lookups if they are running on a multi-hop routing overlay.

In this paper we present Pastel, an extension of Pastry that bridges the gap between the two types of overlays. Pastel has a richer interface than the one currently supported in existing peer-to-peer overlays. This interface enables the use of short lookup paths (or, even better, full information lookups) for direct contact applications, and long, proximity-based lookup paths for routing applications.

The routing tables in a Pastel node can be functionally divided in two parts: a structured part that resembles a Pastry routing table [21] which is used by routing applications, and an unstructured part that maintains a full membership information table with a large number of entries, to support efficient lookups for storage applications.

This extended routing state will allow us to maintain the Pastry functionality that is used by routing applications, and to extend it with highly efficient (i.e., low latency) direct contact for the remaining applications. But the reduced latency is not the only advantage of Pastel. As we will show, there are synergies between the two parts of the routing state that improve the maintenance protocols.

The design of Pastel also raised interesting issues like how to control the extra bandwidth required to maintain a full membership information at each node. We introduce a distinction between strong links (that are aggressively kept up-to-date) and weak links (that have a delayed reponse to unreachability) for the routing state in Pastel. We also show how applications can deal with the presence of weak links without affecting its correctness or performance.

We implemented Pastel in a discrete-event packet level simulator called p2psim [16], and we measured the efficiency of direct contact lookups and the maintenance overhead introduced by the full membership information in Pastel.

Our results show that Pastel can achieve lookups that perform better than Pastry for direct contact applications: in the majority of the cases, multi-hop routing is not required by these applications. Our results also show that the bandwidth required to maintain the extra state is modest, even for a system with thousands of nodes. Furthermore, we achieve this without sacrificing the multi-hop routing interface required by routing applications.

The remainder of the paper is organized as follows. Section 2 gives an overview of Pastry, the system we extended to build Pastel. Section 3 presents an overview of our system. Section 4 presents the Pastel design in detail. Section 5 shows an experimental evaluation of our implementation. Section 6 presents related work, and we conclude in Section 7.

2 Pastry

In this section we summarize the design of Pastry [21], a peer-to-peer overlay that we used as a starting point for the design of Pastel.

Each Pastry node is assigned a 128-bit identifier, and the identifiers are ordered in a circular identifier space modulo 2^{128}. Node ids are uniformly distributed (e.g., they can be computed using a secure hash of the nodes public key or IP address).

Pastry applications use items to partition their workload, and assign to each item a *responsible* node in the overlay, which is the numerically closest live node (assuming that items have ids in the same id space as nodes).

Pastry offers a *route* primitive that, given a message and a key, reliably routes the message to the responsible node. Assuming a Pastry network consisting of N nodes, Pastry can route to any node in less than $\lceil log_{2^b} N \rceil$ hops on average, where b is a system parameter with typical value 4.

The routing algorithm "sees" Pastry ids as a sequence of digits with base 2^b. Each node has a routing table that is organized into $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. Each entry in row n contains the network address of a node whose node id matches the present nodes node id in the first n digits, but whose $n + 1$st digit has one of the $2^b - 1$ possible values other than the present node's $n + 1$st digit (if such a node is found in the overlay). Each entry in the routing table refers to one of potentially many nodes whose node id have the appropriate prefix. Among such nodes, the one closest to the present

node (according to a proximity metric such as the round trip time) is chosen.

In addition to the routing table, each node maintains network addresses for the nodes in its *leaf set*, which consists of the nodes with the $l/2$ numerically closest node ids in each direction of the id space.

Using this state, the *route* operation works recursively as follows. In each step, the node forwards the message to a node whose node id shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current node id. If no such node is found in the routing table, then the message is forwarded to a node whose node id shares a prefix with the key as long as the current node, but is numerically closer to the key than the current node id.

Node joins and departures are handled as follows. When a node *i* joins the overlay, it initializes its state by contacting an existing node asking to route a special message to the id of the joining node, resulting in some responsible node *j*. Then *i* obtains the leaf set from *j*, and the *n*th row of the routing table from the *n*th node encountered along the route to *j*.

To handle node departures, nodes that are neighbors in the id space periodically exchange keep-alive messages. If a node is unreachable for some duration then all nodes in the leaf set are notified of that fact by the node that detected it, and nodes remove this entry from their leaf set. Routing table entries are repaired lazily when an attempt to route through that node fails.

3 Pastel System Overview

We consider the Pastry system organization where nodes are assigned a random 128-bit identifier, and the identifiers are ordered in a circular identifier space modulo 2^{128}.

As in Pastry, we assume that applications use items to partition their workload, and assign to each item a *responsible* node in the overlay, which is the numerically closest live node. However, we also need to take into account a variant of this occurrence when the application uses replication, in which case items have a set of responsible replicas, which we will consider to be the set of *k* nodes whose ids are numerically closer to the item id.

To maintain the routing capabilities of the overlay, and enhance the performance of direct contact applications, we extend Pastry to maintain, side by side, two sets of routing state: The Pastry leaf set and routing table (parameterized by the number of bits in a digit, *b*), and a full information table.

The additional routing information will enable an extended API that satisfies both direct contact applications and applications that relie on structured routing.

We envision that applications with mixed requirements are those that will benefit the most from having these two styles of routing available in the same overlay. Unicast, anycast, multicast and broadcast are all well supported, and are usage patterns shared by many applications. For example, a file sharing system may use broadcast to perform complex queries on shared files, unicast to gauge the responsible for a file that is already identified, and multicast groups to manage groups of nodes sharing and downloading the same files.

4 Pastel Design

In this section we present a design for the Pastel system in more detail.

4.1 Interface

With the two classes of applications in mind, we devised the following application programming interface, which we briefly outline. Note that the presented interface is slightly simplified for clarity. The extension to Pastry consists of the `send` and `broadcast` primitives.

Initialization

`init(node)` – allows the local node to either join an existing Pastel overlay network, by referencing an existing node, or to bootstrap its own, initializing all relevant state.

Message sending

`send(msg, key[, k])` - tries to send the given message directly (i.e., using the full membership information) to the live node with identifier numerically closest to key; if k is specified the k closest nodes are contacted instead.

`route(msg, key)` - routes the message through the structured overlay to the live node closest to key.

`broadcast(msg[, depth])` - broadcasts the message through the structured overlay; if depth is supplied only up to $2^{b^{depth}}$ nodes uniformly spread in the identifier space are reached, otherwise all nodes in the system are to be reached.

Message reception

`deliver(msg, key)` – callback invoked when a message is received and the local node is the recipient for the message, that is, its identifier is numerically closest to key among all live nodes for `route`, or one of k nearby nodes for `send`.

`forward(msg, key, next)` – called when the local node is about to forward the message, whose recipient is the node closest to key, through the node whose identifier is next.

Other operations

`leafs(set)` – callback used when membership changes in the node's leaf set.

`part` – abandon the overlay permanently and in an orderly fashion.

4.2 Node State

Each Pastel node maintains a leaf set, a routing table, and a full information table.

The leaf set and the routing table are identical to the ones implemented in Pastry, and the system tries to populate these with reachable nodes, thus they must be kept current rather aggressively, especially the leaf set (see [21] for a precise description of these tables).

The full information table does not need to be as aggressively kept current, and we even deliberately allow unreachable nodes to remain in this table for some time period (we call these entries weak links). This is because the correctness and liveness of Pastel and the applications that use it do not depend on the freshness of the information present in this table. (The problem of performing lookups using weak links is addressed in the next section.) This table replaces Pastry's neighborhood set with data about all nodes instead of just a sample of nearby nodes. For each entry, we maintain its node identifier, network address, and freshness (time when last contacted).

The storage cost of this data structure is acceptable, especially if secondary storage is considered. For instance, for 1 million 128-bit node IDs and IPv4 addresses only about 25MB are needed. Given this potential size, this table may have to be implemented using a disk friendly data structure such as a B-tree.

Bandwidth costs might be an issue, since nodes can have short sessions [1, 23] and this leads to a large number of notifications about routing information being sent to everyone in the system. However, weak links do not generate much maintenance traffic. This is because we employ a strategy of a delayed response to unreachability: we wait for a certain amount of time T before we remove unreachable nodes from the full information table. Therefore, we do not trigger events due to temporary disconnections. The importance of this strategy is substantiated by experimental studies that have shown that despite short sessions, nodes in peer-to-peer overlays tend to have much longer membership lifetimes [1, 20], or, in other words, when they disconnect from the system, they tend to reconnect later on. Furthermore, weak links also will allow for piggybacking of information about membership changes, which reduces protocol overheads.

Having a full information table is advantageous to the maintenance of structured routing table entries, since, when replacing entries in the routing table, we can choose the best nodes among all live ones, according to some proximity metric. To allow sharing of proximity information, we can associate synthetic coordinates (as determined by a system like Vivaldi [6]) with node entries, instead of round trip times which have to be determined individually. This may lead to better choices for routing table entries than Pastry, that relies on very scarce information and the external knowledge of close by nodes to find neighbors.

In the next sections we show how the full information tables also benefit from their structured counterparts.

4.3 Routing

Pastel routes different message types in different ways, supporting the requirements of diverse applications.

Messages sent with `route` are forwarded through

the structured overlay, using Pastry's routing algorithm [21] (i.e., using only strong links in the routing table and the leaf set).

Messages dispatched with the `send` primitive are routed to the known node closest to the message key, using the full information tables. If the closest node is unavailable (which is more likely than when routing, since `send` uses weak links) the next closest node is tried, and ultimately the live routing table entries are used.

On the other hand, and if k is specified, the message is sent in parallel to the k closest nodes found in the full information table, and the failure of some of them is silently ignored.

We envision that direct routing applications will use the `send` primitive in two distinct ways, depending on the application design.

The first scenario occurs if the application replicates data among the leaf set of the responsible node. In this case the application can use `send` with a replication factor $k > 1$. Since `send` uses weak links, it may occur that the k replicas contacted are not the exact current neighbors of the responsible node, since some nodes may be down, and other new nodes may not be contacted. However, we expect that the replication provided by the application should be enough to tolerate such inconsistencies (which we can see as being similar to node failures).

The other scenario is when the application just intends to contact the exact responsible (and not just one of the replicas). In this case, the `send` primitive will not specify k, but it may fail to contact the responsible node (e.g., if a new responsible as recently joined and the full information table did not reflect this). To address this, the node that is contacted by `send` forwards the message using the same procedure (since it knows nodes closer to the key). Note that even if this indirection is required, this is likely to be faster than structured routing. In the worst case, if `send` fails to reach the responsible node, we can fall back to the traditional `route` primitive.

Messages distributed with `broadcast` are also sent through the structured overlay using a mechanism similar to constrained flooding [2]. A node wishing to broadcast a message makes each node in its routing table responsible for distributing the message to all nodes that share its prefix. When all the descendants of a given node are contained in its leaf set, the message is delivered directly to them, and routing stops. The depth parameter can be used to control maximum the number of nodes that may receive the message, thus avoiding flooding the network. A broadcast with depth D is accomplished by only broadcasting the message to the first D lines in the routing table, decreasing D at each hop. This way, the 2^{bD} nodes reached are uniformly distributed in the identifier space, while at the same time being amongst the nodes closest to the originating node, according to the proximity metric.

4.4 Join and Reconnect Protocols

When a node arrives, it must initialize all relevant state and let others know he joined.

The first steps of the join protocol are very similar to Pastry's: The incoming node asks a known overlay member to route to the id of the incoming node, and the contents of the structured tables (routing table and leaf set) of the intermediate nodes that are contacted are used to initialize the structured tables of the incoming node.

At this point, the full information table is now either empty, if the node is joining the overlay, or possibly outdated, if the node is reconnecting. Also, the system nodes do not know of the existence of the joining node.

Although this does not affect the system's correctness, it affects the performance of the joining node, and the latency of direct queries that should reach the joining node. To address this, we need to disseminate the join information to the remaining system nodes, and the joining node must gather the full membership information.

The protocol for disseminating the information about the node join closely mimics the `broadcast` primitive. Routing table entries are contacted and made responsible for informing nodes that share their prefixes about the join. A node should, however, locally terminate the routing of such messages if it determines the joining node is already present in his full information table. Also, when a node finally delivers the join messages to nodes in its leaf set

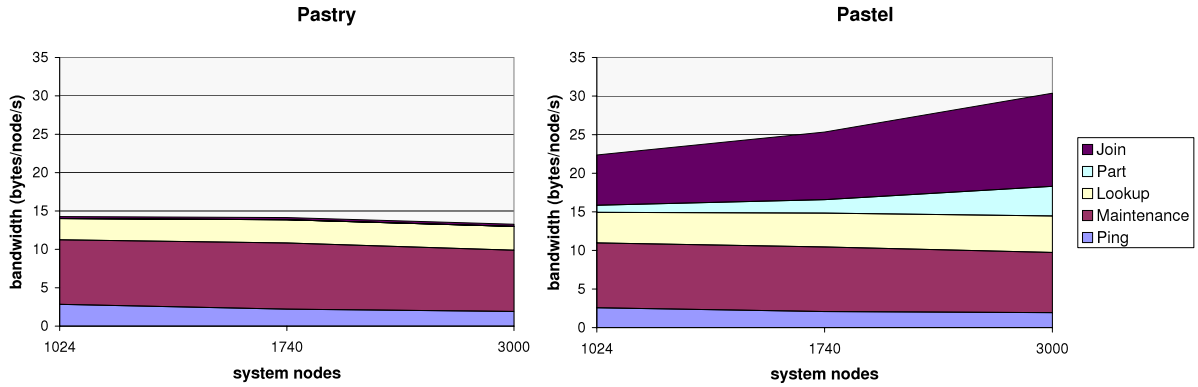

Figure 1: Bandwidth of Pastry and Pastel, with $b = 4$, $L = 16$ and $k = 9$ ($T = 1h$)

(as per the `broadcast` protocol), or when it decides to terminate further routing, it should report back to the joining node. This report contains information about all nodes that are leaves in the broadcast tree, and allows the joining node to gather complete membership information incrementally.

Reconnecting nodes must only download the updates that occurred while they were offline. This can be done efficiently using Merkle trees [14] to determine the missing information.

Note that, since correction is guaranteed by leaf sets and the normal routing mechanisms, all these messages are non-urgent, and can be piggybacked on Pastry maintenance traffic to reduce IP level overhead.

4.5 Leave and Part Protocols

When a node leaves the system permanently (e.g., when the Pastel application is uninstalled) it should use the `broadcast` primitive to transmit a special part message. Again, this message is not urgent and can be piggybacked on Pastry maintenance traffic.

When a node fails without warning, or otherwise transiently abandons the system, this is quickly noted by the members of its leaf set who will evict this node from that set. These nodes then start a timer with time T plus some small random value. If the node shows no activity before the first of these timers expires, the node whose timer expired first will broadcast the special part message on behalf of the departed node.

Upon receiving a part message for a given node, all information about this node should be deleted, including information present in the full information table, and pending timers related to that node.

Again, we stress that this leads to a significant fraction of unreachable nodes in the full information table of any give node, but we leave the solution of this problem to the higher level layers, using replication (and waiting for a reply from any replica) or by redirecting `send` requests to another nearby neighbor.

5 Evaluation

This section demonstrates the benefits of using Pastel through simulation. It shows that the maintenance bandwidth required by Pastel is modest, even for systems with a few thousand nodes, and that we can improve significantly on Pastry's lookup performance. We also point out an improvement that can lead to even better lookup performance in Pastel.

5.1 Experimental Setup

This evaluation uses an implementation of Pastel in p2psim [16], a discrete-event packet level simulator targeted at the evaluation of peer-to-peer protocols.

The simulations ran for 4 hours of simulated time, of which only the last 3 hours are recorded to allow for the system to stabilize. The simulator periodically generates join, part and lookup events, that follow exponential distributions. The average time between joins and parts is 1 hour, and the average time between lookups is 1 minute.

We used three topologies - with 1024, 1740 and

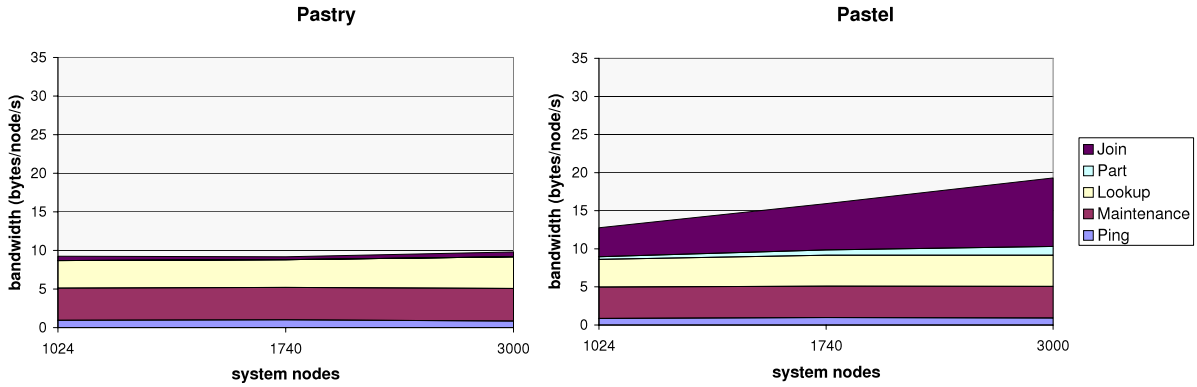

Figure 2: Bandwidth of Pastry and Pastel, with $b = 2$, $L = 8$ and $k = 5$ ($T = 1h$)

3000 nodes. The 1740 nodes topology models a real network as described in [9]. The 1024 nodes topology corresponds to a sample of nodes taken from the 1740 nodes topology, and the 3000 nodes topology is a random euclidean topology designed to approximate the other two in terms of the average round trip time.

5.2 Bandwidth

Our first simulations compare Pastel with Pastry in terms of bandwidth. We compared the average bandwidth consumed by maintenance traffic by every node in the system during the interval when measurements were taken.

Figure 1 shows that Pastel adds some traffic to manage membership changes. This traffic grows linearly with N, which is what can be expected in a system that keeps full membership information. However, even for a system with thousands of nodes, this bandwidth consumption is quite modest, reaching a total of only 30 bytes/second/node in a 3000 node system. This shows that our design is well-suited for systems of this order of participation, and will probably scale well to systems with tens of thousands of nodes (or even a few hundreds of thousands).

In Figure 2 we see similar results for a smaller b (the parameter that defines the base of the digits used in the routing protocol). We can see that bandwidth costs in general are reduced. Reducing b does not negatively impact Pastel's lookup performance due to its full information system, and it can be argued that a smaller b can actually be benefic for applications that take advantage of longer multi-hop paths to distribute load.

5.3 Lookups

The next set of experiments examine the efficiency of full information lookups in Pastel. The gathered data corresponds to the anycast to any one replica: the lookup message is sent in parallel to all the replicas, and the first good answer is accepted.

Figure 3 shows that the percentage of lookups satisfied in the first hop is between 50 and 60%. This already represents an improvement over Pastry, that for the same network sizes and with $b = 3$ requires an average of 3 hops for each lookup. Also note that in Pastel, when a one hop lookup fails, the algorithm falls back to a Pastry lookup. Therefore in the remainder of the cases we can get a performance that is comparable to Pastry.

However, we believe we can improve Pastel further to get the ratio of successful one hop lookups above 90%, a result that was achieved by the One Hop overlay [10], which uses a set of static trees to distribute all join and leave events.

To understand the causes of failed lookups, we investigated how full membership information was being gathered at each node. We evaluated the "quality" of full information tables by measuring how many of those nodes are still live, and how representative are those live nodes of all the live nodes in the system. As can be seen in Figure 4, the percentage of live nodes in the full information ta-

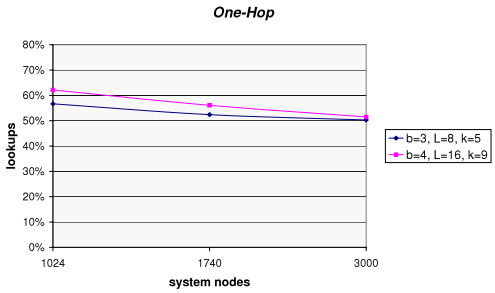

Figure 3: Percentage of lookups satisfied in one hop, with $L = 2^b$ and $k = L/2 + 1$ ($T = 1h$)

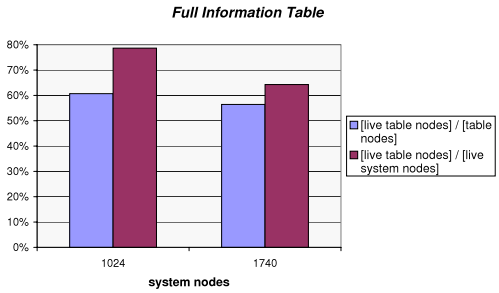

Figure 4: "Quality" of the full information tables, with $T = 1h$ ($b = 4$, $L = 16$ and $k = 9$)

bles remains approximately constant with network growth. On the other hand, the representativity of those nodes when compared to the entire system reduces with network growth. This explains why lookups are less efficient on larger networks.

Figure 4 can also be read the following way. The first column represents the probability of a given node in our full information tables being alive when we try to contact him; the second column represents the probability that given a random identifier we know the node responsible for it.

5.4 Delayed Response to Unreachability

To understand the impact of having a delayed response to unreachability (i.e., waiting before we declare an unreachable node to be removed from the system) on the quality of the full information table we tried halving the time T that we wait before announcing the departure of a node. As Figure 5 shows, varying T contributes to improve the percentage of live nodes in tables (the first column). On the other hand, it also contributes to lower the ratio of the second column, which makes sense since nodes evicted from the full information table may return.

5.5 Replication Factors

We tried to analyze the effect of increasing replication factors in the quality of lookups. In Figure 6 we see that increasing k gains us a little under 10% on the ratio of successful one hop lookups. However, given the large portion of unknown nodes, there is a limit to how successful this can be.

From this analysis we gather that the best way to improve lookup efficiency is to improve the way joins are handled, which will cause less nodes to be unknown to other nodes. In particular, the broadcast primitive we used to disseminate joins did not make sure messages were delivered to all live nodes (this limitation was also noted by other authors [5]). We are currently working on improving Pastel to use a more reliable version of the broadcast primitive, and we believe that the final version of this paper will include results of this improved implementation, probably with even better lookup performance.

6 Related Work

Our work builds on existing routing protocols for peer-to-peer overlays. Initial proposals (like Chord [24], Pastry [21], Tapestry [11], or CAN [18]) require multiple routing hops (typically $O(log\ N)$) to perform a lookup. More recent proposals (like One Hop [10], Beehive [17], or Accordion [13]) have tried to increase the amount of routing state maintained by each node to improve lookup performance. The side effect of these proposals is that the new overlays are not well-suited for applications like peer-to-peer multicast [3, 26] or anycast [4] that exploit the topology formed by the multi-hop lookup paths. Pastel improves on the former group of overlays by achieving a better lookup performance of a single hop in most lookups. Furthermore, Pastel improves on the more recent proposals with increased routing state by also supporting $O(log\ N)$ lookups that enable the deployment of "routing" applications like multicast. Furthermore, we improve on the other algorithm that provides one hop lookups [10] since OneHop uses a set of static distribution trees to disseminate join and leave

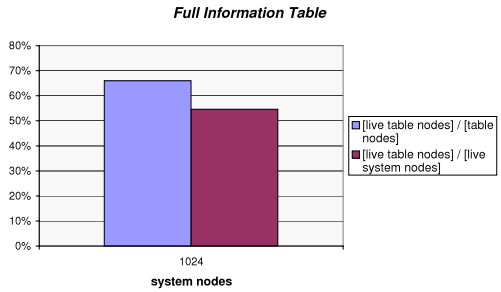

Figure 5: "Quality" of the full information tables, with $T = 30min$ ($b = 4$, $L = 16$ and $k = 9$)

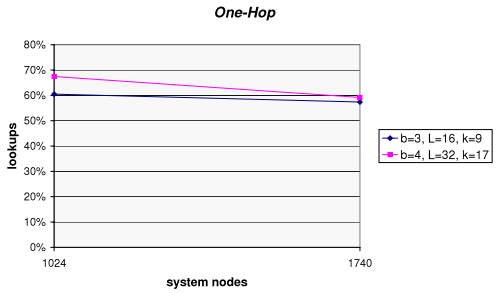

Figure 6: Percentage of lookups satisfied in one hop, with $L = 2 \cdot 2^b$ and $k = L/2 + 1$ ($T = 1h$)

events, which leads to load imbalance at the nodes that are closer to the roots. Our proposal for using broadcast distributes the load uniformly among the system nodes.

Accordion [13] proposes a system that adapts the size of its routing state in order to trade bandwidth for lookup efficiency. Our system proposes a slightly different tradeoff to achieve reasonable bandwidth consumption. The tradeoff in Pastel is between bandwidth consumption and the freshness of the entries in the full information routing table (and, consequently, the percentage of lookups that complete in a single hop).

Pastel introduces an API that gives applications the flexibility to choose between multi-hop and single-hop routing. This extends existing APIs that only support a single "route" primitive. Most notably, a uniformed key-based routing API was proposed [8] with the goals of facilitating independent innovation in overlay protocols, services, and applications, allowing direct experimental comparisons, and encouraging application development by third parties.

7 Conclusion

This paper presents Pastel, a truly generic peer-to-peer overlay that can be used by many applications. Pastel extends Pastry in order to create a substrate that can support both fast, few-hop lookups, for direct contact applications, and slower, multi-hop lookups, for applications that exploit the overlay topology.

Pastel's design demonstrates that synergies exist between the maintenance of full membership information and structured routing state, and that, by exploiting them, bandwidth and storage costs can be kept low, even for reasonably sized and dynamic systems.

We implemented Pastel in a discrete-event packet level simulator and our results show that Pastel has lookups that are more efficient than Pastry's in the majority of the time. Furthermore, the bandwidth required by Pastel is modest, even for systems with thousands of nodes.

Currently, we are working on improving the broadcast primitive used by Pastel's protocols to improve the coverage of our event dissemination, and consequently the percentage of successful one-hop lookups.

References

[1] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, Feb. 2003.

[2] M. Castro, M. Costa, and A. Rowstron. Should we build Gnutella on a structured overlay? In *Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, MA, USA, Nov. 2003.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application-level anycast for highly dynamic groups. In *Networked Group Communication (NGC)*, Munich, Germany, Sept. 2003.

[5] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1510–1520, San Francisco, CA, USA, Apr. 2003.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, Aug. 2004.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, Oct. 2001.

[8] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Feb. 2003.

[9] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. *SIGCOMM Computer Commun. Review*, 32(3), 2002.

[10] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, USA, Mar. 2004.

[11] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 41–52, 2002.

[12] D. R. Karger and M. Ruhl. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, California, USA, Feb. 2004.

[13] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, USA, May 2005.

[14] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology*, number 293 in Lecture Notes in Computer Science, pages 369–378. Springer-Verlag, 1987.

[15] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, USA, Dec. 2002.

[16] p2psim: a simulator for peer-to-peer (p2p) protocols. `http://pdos.csail.mit.edu/p2psim/`.

[17] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 99–112, San Francisco, California, USA, Mar. 2004.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, Aug. 2001.

[19] R. Rodrigues and C. Blake. When multi-hop peer-to-peer routing matters. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, USA, February 2004.

[20] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, USA, Feb. 2005.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[22] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, Oct. 2001.

[23] S. Saroiu, P. K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking 2002 (MMCN)*, Jan. 2002.

[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 *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, Aug. 2001.

[25] J. Stribling, I. G. Councill, J. Li, M. F. Kaashoek, D. R. Karger, R. Morris, and S. Shenker. Overcite: A cooperative digital research library. In *Proc. Fourth International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Feb. 2005.

[26] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.