Peer-to-peer overlays: structured, unstructured, or both?

Miguel Castro, Manuel Costa and Antony Rowstron
Microsoft Research, Cambridge, CB3 0FB, UK

Technical Report
MSR-TR-2004-73

We compare structured and unstructured overlays and derive a hybrid overlay that can outperform both. Unstructured overlays build a random graph and use flooding or random walks on that graph to discover data stored by overlay nodes. Structured overlays assign keys to data items and build a graph that maps each key to the node that stores the corresponding data. Unstructured overlays are widely used in popular applications because they can perform complex queries more efficiently than structured overlays. It is also commonly believed that structured graphs are more expensive to maintain than unstructured graphs and that the constraints imposed by the structure make it harder to exploit heterogeneity to improve scalability. This is not a fundamental problem. We describe techniques that exploit structure to achieve low maintenance overhead, and we present a modified proximity neighbor selection algorithm that can exploit heterogeneity effectively. We performed detailed comparisons of structured and unstructured graphs using simulations driven by real-world traces. Inspired by these results, we developed a hybrid system that uses the graph from structured overlays with the data placement and search strategies of unstructured overlays. The results show that our hybrid system supports complex queries more efficiently than unstructured overlays in realistic scenarios.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

There has been much interest in peer-to-peer data sharing and content distribution applications. They are used by millions of users and they represent a large fraction of the traffic in the Internet [29]. These applications are built on top of large-scale network overlays that provide mechanisms to discover data stored by overlay nodes. There are proposals for two types of overlays: unstructured and structured. This paper presents a detailed comparison of structured and unstructured overlays, and derives a hybrid overlay that can outperform both.

Unstructured overlays, for example Gnutella [1], organize nodes into a random graph and use floods or random walks to discover data stored by overlay nodes. Each node visited during a flood or random walk evaluates the query locally on the data items that it stores. This approach supports arbitrarily complex queries and it does not impose any constraints on the node graph or on data placement, for example, each node can choose any other node to be its neighbour in the overlay and it can store the data it owns. But unstructured overlays cannot find rare data items efficiently because this requires visiting a large fraction of overlay nodes. There has been a large amount of work on improving unstructured overlays, for example [10, 14, 22]. The most recent, Gia [10], provides the best performance.

Structured overlays, for example, [24, 31, 27, 34], were developed to improve the performance of data discovery. They impose constraints both on the node graph and on data placement to enable efficient discovery of data. Each data item is identified by a key and nodes are organized into a structured graph that maps each key to a responsible node. The data or a pointer to the data is stored at the node responsible for its key. These constraints provide efficient support for exact-match queries; they enable discovery of a data item given its key in only $O(logN)$ hops with only $O(logN)$ graph neighbours per node. It is possible to support more complex queries by building indices on top of structured overlays but current solutions perform worse than unstructured overlays when retrieving popular items [20].

It is commonly believed that unstructured overlays provide better support for current mass-market data sharing applications than structured overlays because peers are extremely transient and complex queries are important in this application. For example, recent work [10] has argued that unstructured overlays can cope better with churn and heterogeneity, and that they support complex queries for popular data items, which are the most common, more efficiently.

This paper provides a detailed comparison of structured and unstructured overlays. It explores the design space by decoupling overlay graph maintenance from data placement and search mechanisms. Our findings contradict current wisdom.

We show that structured graphs do not have a fundamental problem in coping with churn and that they can exploit heterogeneity effectively to improve scalability. We present a technique that exploits structure to achieve robustness to churn with low maintenance overhead. It eliminates redundant failure detection probes by using structure to partition failure detection responsibility and to locate nodes that need to be informed about failures and new node arrivals. We also describe how to exploit heterogeneity by modifying any proximity neighbour selection algorithm [7, 34, 17] to adapt the topology to match different node capacities.

The paper presents results of detailed comparisons between several representative structured and unstructured graph maintenance algorithms. These results were obtained using simulations driven by real-world traces of node arrivals and departures [28]. The results show that our techniques enable a structured graph to cope with churn and to exploit heterogeneity with a maintenance overhead lower than unstructured graphs in real-world scenarios.

Inspired by these results, we developed a hybrid system that uses the graph from structured overlays with the data placement and data discovery strategies of unstructured overlays. The hybrid system can use either floods or random walks to locate data but it takes advantage of structure to ensure that nodes are visited only once during a query and to control the number of nodes that are visited accurately. Additionally, it provides applications with the option to leverage efficient exact-match queries for some items, for example, rare items.

We also compared the performance of data discovery in the hybrid system and several representative unstructured overlays using simulations. We used a real trace of content distribution across nodes in a deployed peer-to-peer overlay to guide the simulation [13]. The results show that the hybrid system can discover data more often, faster, and with lower overhead in realistic scenarios.

In Section 2 we describe and compare structured and unstructured graph maintenance protocols assuming a homogeneous setting. Section 3 extends the structured graph maintenance protocol to exploit heterogeneity in peers' resources and compares this with unstructured graph maintenance protocols which exploit heterogeneity. Section 4 compares the performance of content discovery using random walks and flooding on both structured and unstructured graphs. Section 5 discusses future work and Section 6 concludes.

2 Coping with churn

Measurement studies of deployed peer-to-peer overlays have observed a high rate of churn [4, 18, 28]; nodes join and leave these overlays constantly. Therefore, peer-to-peer overlays should be able to cope with a high rate of churn; they should be able to ensure a high probability of success when routing a message between a pair of nodes in the overlay.

Can unstructured graphs cope with churn better than structured graphs ?

Both types of graphs can improve robustness to churn at the expense of increased maintenance overhead by increasing the number of neighbours per node and probing them more frequently to detect and replace failed neighbours.

It is believed that maintaining a structured graph in the presence of churn is more expensive than maintaining an unstructured graph because of the constraints on neighbour selection. This section shows that this is not necessarily the case. It is possible to use structure to achieve better robustness with lower maintenance overhead in a structured graph.

Structured overlays also impose constraints on data placement that can result in high overhead under churn. We study structured graphs without these constraints to keep the evaluation independent of any particular application. Data placement constraints do not result in significant overhead in several applications (for example, multicast [8]) and our hybrid system does not constrain data placement at all.

This section describes the implementation of structured and unstructured graph maintenance protocols in an homogeneous setting and compares their performance. The next section explains how to exploit heterogeneity.

2.1 Unstructured graph

We implemented an unstructured graph maintenance protocol based on the specification of Gnutella version 0.4 [16] but we added several optimizations to the protocol to ensure a fair comparison.

Gnutella 0.4 organizes overlay nodes into a random graph. Each node in the overlay maintains a neighbour table with the network addresses of its neighbours in the graph. The neighbour tables are symmetric; if node x has node y in its neighbour table then node y has node x in its neighbour table. There is an upper and lower bound on the number of entries in each node's neighbour table.

A joining node uses a random walk starting from a *bootstrap* node, which is randomly chosen from the set of nodes already in the graph, to find other nodes to fill its neighbour table. It sends the bootstrap node a *neighbour discovery* message with a counter that is initialized to the number of nodes required to fill its neighbour table. Upon receiving a discovery message, a node checks whether it has less neighbours than the upper bound. If this is the case, the node sends a message to the joining node inviting it to become a neighbour and decrements the counter in the neighbour discovery message. In either case, the neighbour discovery message is forwarded to a randomly chosen neighbour if the counter is still greater than zero. To increase resilience to node and network failures, all neighbour discovery messages are acknowledged. If a node does not receive an acknowledgement within a timeout, it selects another neighbour at random and forwards the neighbour discovery message to that neighbour.

In addition to joins, nodes need to detect failures and replace faulty neighbours. Every t seconds each node sends an *I'm alive* message to every node in its neighbour table. Since all nodes do the same and neighbour tables are symmetric, each node should receive a message from each neighbour in each t seconds. If a node does not receive a message from a neighbour, it explicitly probes them and if no reply is received the node is assumed to be faulty. We used $t = 30$ seconds in this paper. Nodes maintain a cache of other nodes that they use to replace failed neighbours. If the cache is empty, they obtain new neighbours by sending a neighbour discovery message to a randomly chosen neighbour. All messages sent between the nodes are used to replace explicit *I'm alive* messages.

Simulation results show that this protocol does not produce sufficiently random graphs in the presence of churn, which leads to poor query performance. This happens because the neighbour table of a joining node and those of its neighbours are likely to share some nodes, creating a clustering effect. We overcome this problem by forwarding the neighbour discovery message over a number of random hops after each neighbour invitation is sent. We add a hop counter to discovery messages that is set to R by every node that replies with a neighbour invitation. Nodes decrement the hop counter when they forward a discovery message and they only consider sending a neighbour invitation when the counter is less than or equal to zero. We used $R = 5$ in this paper.

We use unbiased random walks because we found that biasing the random walk to nodes with low degree reduces overhead but results in poor query performance. We also experimented with flooding of discovery messages (as specified in the Gnutella 0.4 protocol) but this results in additional overhead without improved robustness or query performance.

2.2 Structured graph

There are several structured graph maintenance protocols. We chose an implementation of Pastry [27] called MS Pastry [23, 5] because it has good performance under churn and it has an efficient implementation of proximity neighbour selection [7] that we modified to exploit heterogeneity (as described in the next section).

Structured overlays map keys to overlay nodes. Overlay nodes are assigned *nodeIds* selected from a large identifier space and application objects are identified by keys selected from the same identifier space. Pastry selects nodeIds and keys uniformly at random from the set of 128-bit unsigned integers and it maps a key k to the node whose identifier is numerically closest to k modulo 2^{128}. This node is called the key's root. Given a message and a destination key, Pastry routes the message to the key's root node. Each node maintains a routing table and a leaf set to route messages.

NodeIds and keys are interpreted as a sequence of digits in base 2^b. We use $b = 1$ in this paper. The routing table is a matrix with $128/b$ rows and 2^b columns. The entry in row r and column c of the routing table contains a random nodeId that shares the first r digits with the local node's nodeId, and has the $(r+1)$th digit equal to c. If there is no such nodeId, the entry is left empty. The uniform random distribution of nodeIds ensures that only $log_{2^b}N$ rows have non-empty entries on average. Additionally, the column in row r corresponding to the value of the $(r+1)$th digit of the local node's nodeId remains empty.

Nodes use a *neighbour selection function* to select between two candidates for the same routing table slot. Given two candidates y and z for slot (r, c) in node x's routing table, x selects z if z's nodeId is numerically closer than y's to the nodeId obtained by replacing the $(r+1)$th digit of x's nodeId by c. This neighbour selection function promotes stability in routing tables while distributing load. We chose not to use proximity neighbour selection because it increases overhead slightly and low delay routes do not seem important for the applications we study in this paper.

The leaf set contains the $l/2$ closest nodeIds clockwise from the local nodeId and the $l/2$ closest nodeIds counter clockwise. The leaf set ensures reliable message delivery. We use $l = 32$ in this paper, which provides high robustness to large scale failures and high churn rates.

At each routing step, the local node normally forwards the message to a node whose nodeId shares a prefix with the key that is at least one digit longer than the prefix that the key shares with the local node's nodeId. If no such node is known, the message is forwarded to a node whose nodeId is numerically closer to the key and shares a prefix with the key at least as long. The leaf set is used to determine the destination node in the last hop.

Exploiting structure to reduce maintenance overhead Structured overlays can use structure to reduce maintenance overhead in several ways. First, several structured overlays use structure to initialize the routing tables of joining nodes efficiently and to announce their arrival.

Node joining in Pastry exploits the graph structure as follows. A joining node x picks a random nodeId X and asks a bootstrap node a to route a special join message using X as the destination key. This message is routed to the node z with nodeId numerically closest to X. The nodes along the overlay route add routing table rows to the message; node x obtains the rth row of its routing table from the node encountered along the route whose nodeId matches x's in the first $r-1$ digits and its leaf set from z. After initializing its routing table, x sends the rth row of the table to each node in that row. This serves both to announce x's presence and to gossip information about nodes that joined previously. Each node that receives a row considers using the new nodes to replace entries in its routing table.

Additionally, structured overlays can eliminate redundant failure detection probes by using structure to partition failure detection responsibility and to locate nodes that need to be informed when a failure is detected. For example, MS Pastry uses this technique to reduce the number of liveness probes in the leaf set by a factor of 32. Each node sends a single *I'm alive* message every t_l seconds to its left neighbour in the id space. If a node does not receive a message from its right neighbour, it probes the neighbour and marks it faulty if it does not reply. When it marks the neighbour faulty, it discovers the new member of its leaf set by querying the right neighbour of the failed node and informs all the members of the new leaf set about the failed node. If several consecutive nodes in the ring fail, the left neighbor of the leftmost node will detect the failure and repair provided the number of consecutive nodes that failed is less than $l/2 - 1$. We use $t_l = 30$ seconds in this paper, which is equal to the period between *I'm alive* messages in the unstructured graphs.

The technique can be extended to eliminate fault detection probes sent to routing table entries. This can be done in routing tables that constrain each node x to point to nodes whose identifiers are the closest to specific points in the identifier space derived from x's nodeId, for example, the original Chord [31] finger table and Pastry's constrained routing table [6]. For example, Pastry's constrained routing table enables a node that detects the failure of its right neighbour to locate all nodes with routing table entries pointing to the failed node with an expected cost of O(log N) messages. We chose not to use the constrained routing table because it eliminates the flexibility necessary to cope with heterogeneous peers as described in the next section.

MS Pastry uses a different strategy to detect failures in the routing table. Since the routing table is not symmetrical, a node explicitly probes every member every t_r seconds to detect failures. The routing table probing period t_r is set dynamically by each node based on the node failure rate in the overlay observed by the node [5]. We configured MS Pastry to achieve a 1% loss rate, i.e., a message routed between a pair of nodes has a probability of 99% of reaching the destination even in the absence of retransmissions.

Pastry also has a *periodic routing table maintenance* protocol to repair failed entries. Each node x asks a node in each row of the routing table for the corresponding row in its routing table. x chooses between the new entries in received rows and the entries in its routing table using the neighbour selection function defined above. This is repeated periodically, for example, every 20 minutes in the current implementation. Additionally, Pastry has a *passive routing table repair* protocol: when a routing table slot is found empty during routing, the next hop node is asked to return any entry it may have for that slot.

2.3 Experimental comparison

We compare the maintenance overhead of the different graphs using a packet-level discrete-event simulator. We simulated a transit-stub network topology [33] with 5050 routers. There are 10 transit domains at the top level with an average of 5 routers in each. Each transit router has an average of 10 stub domains attached, and each stub has an average of 10 routers. Routing is performed using the routing policy weights of the topology generator [33]. The simulator models the propagation delay on the physical links. The average delay of router-router links was 40.7ms. In the experiments, each end system node was attached to a randomly selected stub router with a link delay of 1ms.

The simulation is driven using a real-world trace of node arrivals and failures from a measurement study of Gnutella. The study [28] monitored 17,000 unique nodes in the Gnutella overlay over a period of 60 hours. It probed each node every seven minutes to check if it was still part of the overlay. The average session time over the trace was approximately 2.3 hours and the number of active nodes in the overlay varied between 1,300 and 2,700. The failure rate and arrival rates are similar but there are large daily variations (more than a factor of 3). There was no application-level traffic during this experiment to isolate the graph maintenance overhead.

We compare the maintenance overhead of Gnutella 0.4 and Pastry. We used two configurations of Gnutella 0.4: *Gnutella 0.4 (4)* bounds the number of neighbours to be at least 4 and no more than 12, *Gnutella 0.4 (8)* bounds the number of neighbours to be at least 8 and no more than 32. We chose these parameters because *Gnutella 0.4 (4)* has a maintenance overhead lower than Pastry whereas *Gnutella 0.4 (8)* has a higher overhead. Figure 1 shows the maintenance overhead measured as the average number of messages per second per node. The x-axis represents simulation time.

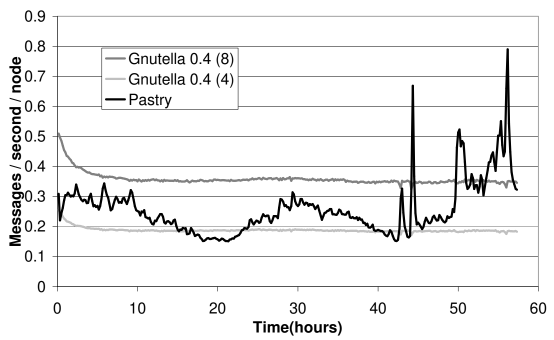

Figure 1: Maintenance overhead in messages per second per node over time for the Gnutella 0.4 and Pastry graphs.

Most of the overhead is due to fault detection messages in the three graphs. In the Gnutella overlay, nodes send *I'm alive* messages to each of their neighbours every 30 seconds. The average number of links per node over the trace is 5.8 in *Gnutella 0.4 (4)* and 11.0 in *Gnutella 0.4 (8)*. Therefore, the expected overhead due to fault detection is 0.19 and 0.37 messages per second per node in *Gnutella 0.4 (4)* and *Gnutella 0.4 (8)*, respectively.

Pastry's maintenance overhead is between the overhead of *Gnutella 0.4 (4)* and *Gnutella 0.4 (8)* most of the time. Pastry is significantly more robust than either of them because it maintains considerably more neighbours. Each node has 32 neighbours in the leaf set alone and it detects their failure as fast as the unstructured graphs detect neighbour failures. A node only gets partitioned from the overlay if 15 nodes fail in Pastry whereas it only takes 6 nodes to fail in *Gnutella 0.4 (4)* and 11 in *Gnutella 0.4 (8)*.

Pastry is able to achieve low maintenance overhead because it exploits structure. The overhead for fault detection of leaf set members is only 0.03 messages per second per node even though there are 32 nodes in each node's leaf set. Additionally, Pastry tunes the routing table probing period to achieve 1% loss rate (using the techniques described in [5]). This ensures that it uses the minimum probe rate that achieves the desired reliability. Pastry's maintenance overhead varies with the failure rate observed during the trace because the self-tuning technique increases the probe rate when the node failure rate increases.

It is important to note that the maintenance overhead is negligible in the three system. For example, the average number of messages per second per node over the trace is only 0.26 in Pastry. Furthermore, the vast majority of these messages are smaller than 100 bytes on the wire. Therefore, the overhead is less than 26 bytes per second, which is negligible even for users with slow dialup connections.

The maintenance overhead is constant in the unstructured graph but grows with N in the structured graph. However, it grows very slowly. The fault detection traffic, which accounts for most of the maintenance overhead, is constant for leaf set members and it is proportional to $log_2(N)$ for routing table entries. For example, increasing N to one billion nodes with a similar pattern of node arrivals and departures would increase maintenance traffic in the structured graph to less than 0.69 messages per second per node (or less than 69 bytes per second per node), which is still negligible.

3 Exploiting heterogeneity

Nodes in deployed peer-to-peer overlays are heterogeneous [28]; they have different bandwidth, storage, and processing capacities. An overlay that ignores the different node capacities must bound the load on any node to be below the load that the least capable nodes are able to sustain; otherwise, it risks congestion collapse. Hence, it is important to exploit heterogeneity to improve scalability.

Can unstructured graphs exploit heterogeneity more effectively than structured graphs ?

Structured graphs have constraints on the graph topology that reduce flexibility to adapt the topology to exploit heterogeneity. However, some structured graphs have significant flexibility in the choice of some overlay neighbours, which is important to implement proximity neighbour selection [34, 27, 17].

These structured graphs can exploit heterogeneity by modifying the proximity neighbour selection algorithm to choose nodes with high capacity as overlay neighbours. We show that this is as effective as recent proposals to adapt unstructured graphs [10].

This section describes the implementation of several structured and unstructured graph maintenance protocols that exploit heterogeneity and compares their performance.

3.1 Unstructured graphs

We implemented two unstructured graph maintenance algorithms that exploit heterogeneity: a version of *Gnutella 0.6* [2] and a version of Gia [10].

Gnutella 0.6 extends the Gnutella 0.4 protocol by adding the concept of super-peers [3]. Nodes that are capable of contributing enough resources to the overlay are classified as super-peers and organized into a random graph using the optimized version of the Gnutella 0.4 protocol (which was described in the previous section). Ordinary nodes are not part of the random graph. Instead, each ordinary node attaches to a small number of randomly selected super-peers and proxies its data discovery queries through them. Ordinary nodes select super-peers to attach to using a random walk with a modified neighbour discovery message and they exchange *I'm alive* messages with the selected super-peers to detect failures. This topology places most of the data discovery and graph maintenance load on super-peers.

Gia [10] provides a more fine-grained adaptation to heterogeneity. Each node selects a numerical *capacity* value that abstracts the amount of resources that it is willing to contribute to the overlay. Gia adapts the graph topology such that nodes with higher capacity have higher degree. Since high-degree nodes receive a larger fraction of the traffic, this ensures that they have the capacity to handle this traffic. Gia's fine-grained approach to exploit heterogeneity can perform better than simply using super-peers [10].

We implemented Gia exactly as described in [10]. Node discovery is implemented using a random walk (as described for Gnutella 0.4) but the nodes use Gia's *pick_neighbor_to_drop* function [10] to decide whether to send back a neighbour invitation message. Topology adaptation is driven by Gia's *satisfaction_level* function, which increases with the sum of the ratio between the capacity and degree of each neighbour. This function is evaluated periodically and nodes with a low satisfaction level attempt to find a new neighbour to increase the level. The adaptation interval is computed as in Gia (with the parameters $K = 256$ and $T = 10$ seconds).

3.2 Structured graphs

We implemented two structured graph maintenance protocols based on Pastry that exploit heterogeneity: *SuperPastry* uses super-peers like Gnutella 0.6 and *HeteroPastry* uses topology adaptation like Gia.

It is simple to exploit the super-peers concept in a structured overlay. The super-peers are organized into a structured graph using the Pastry algorithm described in the previous section. Ordinary peers do not join this graph. Instead they attach to a small number of super-peers as in Gnutella 0.6. Ordinary peers select super-peers to attach to by routing to random destination keys through a bootstrap super-peer. They exchange *I'm alive* messages with the selected super-peers to detect failures as in Gnutella 0.6.

The implementation of capacity-aware topology adaptation in structured graphs is less obvious. We propose a simple solution based on existing proximity neighbour selection algorithms [27, 34, 17]. These algorithms select the closest neighbours in the underlying network subject to the structural constraints on the graph. They can be modified to provide capacity-aware topology adaptation by using a proximity metric that reflects node capacity.

HeteroPastry uses the Pastry algorithm described in the previous section except that it achieves capacity-aware topology adaptation by modifying the neighbour selection function to take node capacity into account. Given two candidates y and z for slot (r, c) in node x's routing table, x selects z if it has capacity greater than y or if z and y have the same capacity and z's nodeId is numerically closer than y's to the nodeId obtained by replacing the $(r + 1)$th digit of x's nodeId by c. We assume that node capacities are quantized into a few discrete values for the randomization based on nodeIds to be effective at distributing load. It is possible to design more complex neighbour selection functions that combine several capacity metrics and even network proximity.

In addition to specifying capacity, nodes can specify an upper bound on their indegree, i.e., the number of nodes with routing table entries pointing to them. This bound is likely to be a function of their capacity. We modified Pastry to ensure that the number of routing table entries pointing to a node does not exceed the specified bound. Each node x keeps track of nodes with routing table entries that point to x (backpointers) and sends *backoff* messages when the number of backpointers exceeds the indegree bound. It is necessary to keep track of backpointers because neighbour links in Pastry routing tables are not symmetric. Neighbour links in the leaf set are symmetric and their number is fixed at 32 in this paper. They are not counted as part of the indegree of x unless they also have a routing table entry pointing to x.

Nodes keep track of backpointers by passively monitoring messages received from other nodes. They add a node to the backpointer set when they receive a message from the node and, every D seconds, they remove nodes from which they did not receive messages for more than $2D$ seconds. D is set to the routing table probing period because nodes send probes to their routing table entries every routing table period.

If the number of backpointers exceeds the bound after adding a new node, the local node x selects one of the backpointers for removal and sends that node a backoff message. For each backpointer y with x in slot (r, c) of its routing table, the numerical distance between x's nodeId and the nodeId obtained by replacing the $(r+1)$th digit of y's nodeId by c is computed. x selects the node with the maximal distance for eviction. This policy is dual of the neighbour selection function (except that it is oblivious to capacity) to provide stability.

Nodes that receive a backoff message remove the sender from their routing tables and insert the sender in a backoff cache. We modified the neighbour selection function to ensure that it never selects nodes in the backoff cache. The current implementation removes entries from the backoff cache after four routing table probing periods.

Our solution is not applicable to some structured graphs that provide no flexibility at all in the selection of neighbours, for example, the original Chord [31] and CAN [24]. It is possible to use virtual nodes [31] to adapt these structured graphs to different node capacities. Each physical node can simulate a number of virtual overlay nodes proportional to its capacity. The problem is that node capacities can vary by several order of magnitude. Therefore, the number of virtual nodes must be much larger than the number of physical nodes, which results in a large increase in maintenance traffic.

3.3 Experimental comparison

We compared the maintenance overhead of the different graph maintenance algorithms that exploit heterogeneity to achieve scalability. We used the experimental setup in Section 2.3, which does not include any query traffic, to isolate the maintenance overheads.

Gnutella 0.6 and SuperPastry were configured with similar parameters to allow a fair comparison. Each ordinary node selected 3 super-peers as proxies and each super-peer acted as a proxy for up to 30 ordinary nodes. Each super-peer in Gnutella 0.6 had at least 10 super-peer neighbours and at most 32. The indegree bound of super-peers in SuperPastry was also 32. The simulator provided each joining node with a randomly selected super-peer to bootstrap the joining process and joining nodes were marked super-peers with a probability of 0.2. Figure 2 shows the maintenance overhead measured as the number of messages sent per second per node.

The maintenance overhead is dominated by the cost of failure detection as before. In Gnutella 0.6, a node has 7.5 neighbours on average, which results in 0.25 *I'm alive* messages per second per node on average. This accounts for most of the control traffic has shown in the figure. Both systems incur the same communication overhead between ordinary peers and super-peers. SuperPastry achieves lower overhead than Gnutella 0.6 because it exploits structure to reduce failure detection overhead.

We also ran experiments to compare the maintenance overhead of Gia and HeteroPastry. Gia was configured using the parameters in [10]. The lower bound on the number of neighbours in Gia is 3 and the upper bound is $max(3, min(128, \frac{C}{4}))$ [10], where C is the capacity of the node. We use the same bounds on the indegree of nodes in HeteroPastry. The capacity of a node (in both overlays) is selected when it joins according to the probabilities in Table 1, which were taken from [10].

Figure 3 plots the maintenance overhead in messages per second per node against time for Gia and HeteroPastry. Failure detection messages account for most of the overhead as in previous experiments. Nodes in Gia have 15.6 neighbours on average, which results in 0.52 *I'm alive* messages per second per node. The overhead of HeteroPastry is almost identical to the overhead incurred by the version of Pastry that does not exploit heterogeneity and does not bound indegrees (which is shown in Figure 1).

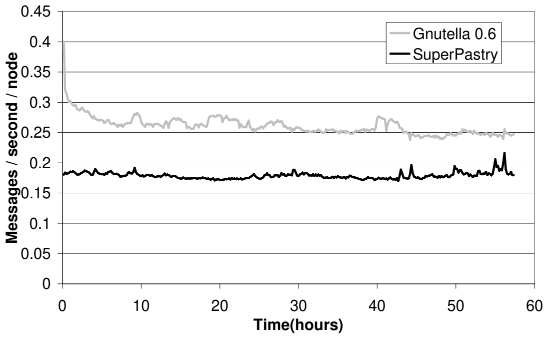

Figure 2: Maintenance overhead in messages per second per node over time for the two graphs using super-peers.

Capacity	Probability
1	0.2
10	0.45
100	0.3
1000	0.049
10000	0.001

Table 1: Node capacity distribution

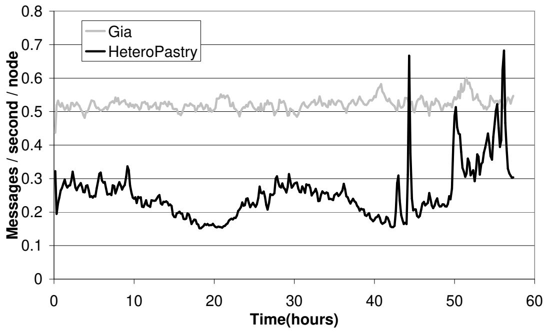

Figure 3: Maintenance overhead in messages per second per node over time for Gia and HeteroPastry.

Figure 3 shows that the overhead of topology adaptation in HeteroPastry is negligible. The next set of results show that this topology adaptation is also effective.

We examined the routing tables of live HeteroPastry nodes five hours into the trace and calculated the average capacity of the nodes in routing table entries at each routing table level across the 2627 live nodes. Figure 4 shows the results.

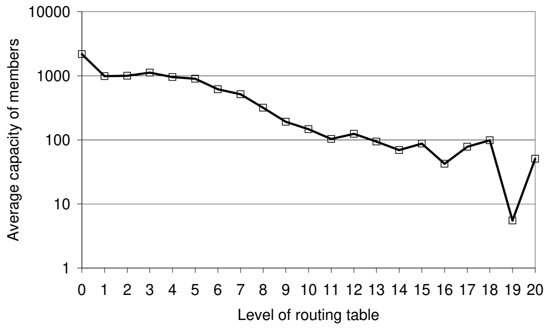

Figure 4: Average capacity of nodes in routing table entries at each level in HeteroPastry.

Topology adaptation fills routing tables with high capacity nodes. The average capacity of nodes in levels up to 5 is above 897. The capacity decreases when the level increases because of stronger structural constraints. A node in level l of the routing table must match the nodeId of the local node in the first l digits. The size of the set of nodes that can be selected to fill slots at level $l + 1$ is half the size of the set of nodes that can fill slots at level l. Therefore, the probability that these sets include high capacity nodes decreases as the level increases. Since most nodes have less than 12 ($log_2(2627)$) levels in their routing tables, there is some noise for levels above 12.

We also measured the average indegree of nodes with each capacity value at the same point in time. The results are in Figure 5. The average indegree of the two nodes with capacity 10000 is above the indegree bound of 128. This happens because nodes are very likely to select nodes with capacity 10000 for the top levels of their routing tables and these pointers are only removed after the node receives a backoff message. The results show that topology adaptation in HeteroPastry is effective at distributing the indegree according to capacity.

4 Discovering data

Complex queries are important in mass-market data sharing applications [10]. Since users do not know the exact names of the files they want to retrieve, the exact-match queries offered by structured overlays are not directly useful in these applications. Users discover data with keyword searches, which are readily supported by unstructured overlays that visit a subset of random nodes in the overlay and execute the search query locally at each visited node.

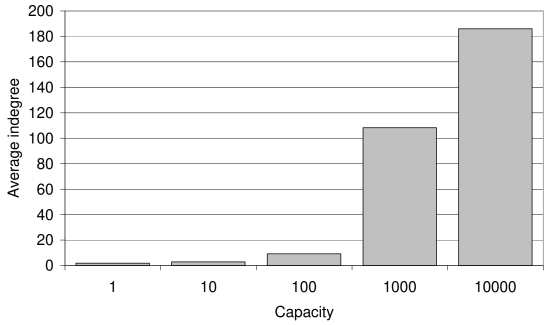

Figure 5: Average indegree of nodes with each capacity value.

Can unstructured graphs support complex queries more efficiently than structured graphs ?

Several research prototypes support keyword searches using the exact-match queries of structured overlays [26, 32, 15, 19] to implement inverted indices. The basic idea is to use the structured graph to map keywords to overlay nodes. The node responsible for a keyword stores an index with the location of all documents that contain the keyword. When a file is added to the system, the nodes responsible for the keywords in the file are contacted to update the appropriate indices. A query for documents containing a set of keywords contacts the nodes responsible for those keywords and intersects their indices.

Unfortunately, this approach has several problems. Maintaining the indices up to date in the presence of churn is expensive and popular keywords may be mapped to low capacity nodes that cannot cope with the load [10]. Additionally, the queries can be expensive because they require computing the intersection of large indices. The analysis in [20] shows that this approach performs worse than flooding queries to 60,000 nodes in a random graph. Therefore, this approach performs significantly worse than recent unstructured overlays like Gia [10]. Additionally, unstructured overlays can support even more sophisticated queries that are not supported by the inverted indices approach, for example, regular expressions and range queries on multiple attributes.

This section explores a different approach to supporting complex queries in structured graphs. We developed a hybrid system that uses the graph from structured overlays with the data placement and data discovery strategies of unstructured overlays. It can support arbitrarily complex queries using either floods or random walks over the structured graph but it takes advantage of structure to ensure that nodes are visited only once during a query and to control the number of nodes that are visited accurately.

The results in the previous sections show that it is possible to maintain a structured graph that exploits heterogeneity with low maintenance overhead. Additionally, the hybrid system does not constrain data placement; nodes do not have to incur the overhead of updating distributed indices for each keyword in their files.

This section compares the performance of random walks and floods on the graphs described in the previous sections.

4.1 Unstructured graphs

We used random walks to discover data because they have been shown to induce lower overhead than the constrained floods [21] used by current versions of Gnutella. These random walks are biased to prefer nodes with higher degree in Gia and are unbiased in the other unstructured graphs. The original Gia [10] biased the random walks to prefer nodes with higher capacity but our experimental results indicate that preferring nodes with higher degree yields both higher success rate and lower delay. We present results for the more efficient variant of Gia.

We observed that random walks in Gia were likely to visit the same node more than once, which resulted in worse search performance. We added a list to each query with all the nodes already visited by the query to prevent this. Nodes do not forward a query to a node that is in this list.

All unstructured graphs use *one hop replication*, which has been shown to improve search performance in unstructured overlays [10]. A node replicates an index of its content at each of its neighbours. In Gnutella 0.6, these indices are only replicated at super peers.

4.2 Structured graphs

The hybrid system exploits structure to implement random walks and constrained floods more efficiently.

Flooding in random graphs is inefficient because each node is likely to be visited more than once. In a graph with an average degree of k, a flood that visits all nodes will send on average $(k-1) \times N$ messages (where N is the size of the overlay). Additionally, it is difficult to control the number of nodes visited during a constrained flood. Floods are constrained using a time-to-live field in the query message that is decremented every time the query is forwarded. The query is not forwarded when the time-to-live field drops to zero. This provides very coarse control over the number of nodes visited.

The hybrid system can do better by replacing flooding with the broadcast mechanisms that have been proposed for structured overlays [25, 9, 12]. We use Pastry's broadcast mechanism [9] to flood queries to overlay nodes. A node y broadcasts a query by sending the query to all the nodes x in its routing table. Each query is tagged with the routing table row r of node x. When a node receives a query tagged with r, it forwards the query to all nodes in its routing table in rows greater than r if any.

A node may have a missing entry in a slot in its routing table, for example, because it pointed to a node that failed. The broadcast overcomes this problem by using Pastry to route the query to a node with the appropriate nodeId to fill the slot (if there is any) [9]. Almost all nodes receive the query only once but the technique to deal with empty routing table slots may result in a small number of duplicates.

We place an upper bound on the row number of entries to which the query is forwarded to constrain the flood. This bounds the number of nodes visited to a power of two. It is simple to extend this mechanism to provide arbitrarily fine grained control over the number of nodes visited.

It is simple to modify this mechanism to perform random walks rather than floods by adding a set of nodes to visit in the query message. A random walk query message includes the tag r, an array q with queues of nodes indexed by routing table row, and a bound d on the maximum row number to traverse. When the query is received at node x, it appends the nodes in each routing table row r' to queue $q[r']$ provided that $r < r' \leq d$. Then, if queue $q[r]$ is not empty, x removes the next node from the queue and forwards the query to this node. If $q[r]$ is empty, the query is forwarded to the first node in queue $q[r+1]$ and r is incremented. If all queues are empty, the random walk is complete.

The results in the previous section show that the average capacity of the nodes in routing table entries in HeteroPastry decreases as the row number increases. Therefore, the mechanism that we use to bound the floods and random walks biases them to visit nodes with higher capacity in HeteroPastry.

We could implement random walks simply by walking along the ring formed by neighbouring nodes in the id space. This is an effective random walk over the content because nodeIds are independent of the content stored by the nodes. However, this implementation does not achieve biased random walks in HeteroPastry.

We also implement *one hop replication* in the hybrid system. Each node replicates an index of its local content on the nodes in its routing table. Therefore, it replicates its index in $log_2(N)$ other nodes. Since routing tables are not symmetric, this number is independent of its indegree; the number of routing table entries pointing to the node could even be zero. The unstructured overlays do not have this flexibility because neighbour tables are symmetric. This flexibility improves the performance of HeteroPastry.

4.3 Experimental comparison

We compared the performance of random walks on structured and unstructured graphs. We used the basic experimental setup described in the previous sections but we simulated queries and node file stores.

We used a real-world trace of files stored by eDonkey [13] peers to model the sets of files stored by simulated nodes. There are 37,000 peers in the trace and, for each peer, there is a record with the identifiers of the files stored by the peer. Figure 6 shows the distribution of the number of files stored by each peer. It excludes the 25,172 peers that have no files. We model the set of files stored by each node as follows: when a node joins, the simulator chooses a random unused record from the trace and assigns the files in the record to the node.

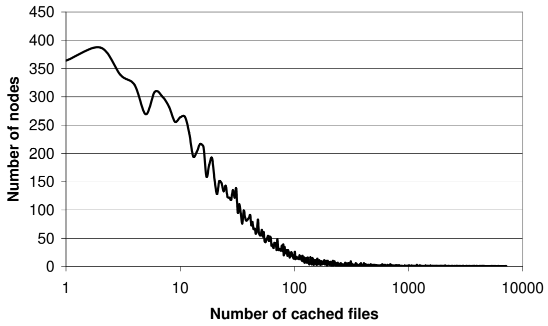

Figure 6: Distribution of the number of files per node for the eDonkey file trace [13].

There are approximately 923,000 unique files. File copies exhibit a heavy-tailed zipf-like distribution as shown in Figure 7. Full details about trace can be found in [13].

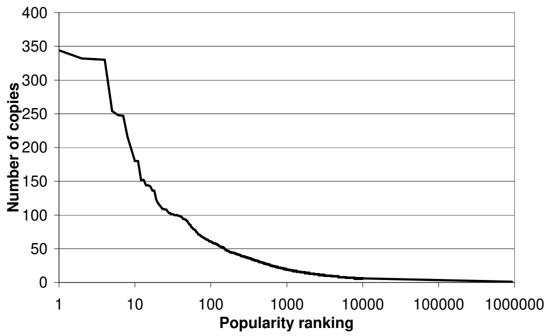

Figure 7: Number of files versus file rank for the eDonkey file trace [13].

The eDonkey trace does not include queries but the number of copies of a file is strongly correlated with the number of queries that it satisfies. Therefore, our query distribution matches the distribution of the number of copies of files.

Each node generates 0.01 query messages per second using a Poisson process and each query searches for a file in the trace. The simulator maintains the distribution of the number of copies of files stored by nodes that are currently in the overlay. The target file for each query is chosen from this distribution (which is a sample of the distribution in 7). This ensures that at least a copy of the target file is stored in the overlay when the query is initiated.

In all the experiments, we bound random walks to visit at most 128 nodes. When a node x receives a query, it checks if the target file is stored locally or if it is stored by nodes whose indices are replicated locally. In the first case, the query is satisfied and x does not forward the query further. In the second case, x contacts a random node y which it believes has a copy of the file. If y has the file, the query is satisfied and y sends an acknowledgment back to x. If x receives the acknowledgment before a timeout, it stops forwarding the query. Otherwise, x contacts another random node that it believes has the file or it forwards the query if there are no more such nodes.

We measured the fraction of queries that are satisfied and the delay from the moment a query is initiated until it is satisfied. We also measured the load by counting the number of messages sent per second per node.

4.3.1 Homogeneous

We first compare the overlays that do not exploit heterogeneity: Gnutella 0.4 and Pastry. Figure 8 shows the number of messages per second per node and Figure 9 shows the success rate of queries.

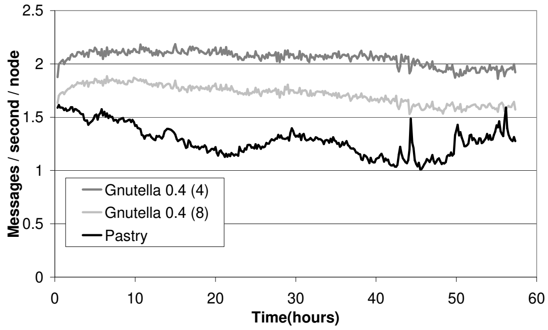

Figure 8: Messages per second per node.

In the absence of queries, Gnutella 0.4 (4) had lower message overhead than Gnutella 0.4 (8) because nodes have less neighbours on average. This is reversed with queries because most failure detection traffic is suppressed by application traffic and more neighbours result in more replicated indices. Gnutella 0.4 (8) has approximately two times more replicas of node indices, which results in a higher success probability and shorter random walks.

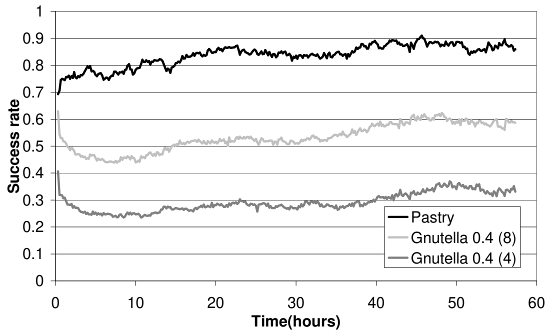

Figure 9: Query success rate.

Pastry's success rate is significantly higher than the success rate of the two Gnutella configurations. This happens because Pastry has 33% more index replicas than Gnutella 0.4 (8) and the distribution of indegrees (and the number of index replicas) is less uniform in Pastry. The nodes with higher indegree are more likely to be visited by random walks, which increases the success rate and reduces average overhead. This imbalance in the distribution of indegree also results in a slight load imbalance; the most loaded Pastry node services twice as many messages per second as the most loaded node in Gnutella 0.4 (8).

Figure 10 shows the average delay for a successful query. Pastry has lower delay for the same reason that its success rate is higher.

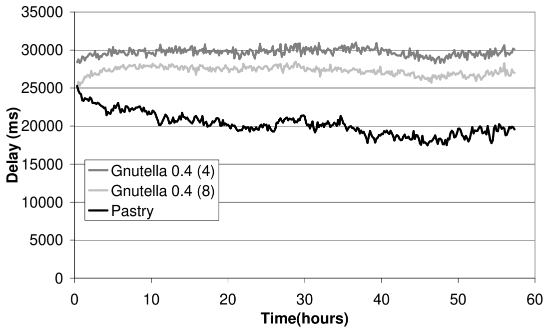

Figure 10: Query delay for successful queries.

4.3.2 Heterogeneous

We compared the performance of data discovery on the graphs that exploit heterogeneity. Figure 11 shows the query success rate, Figure 12 shows the delay for successful queries, and Figure 13 shows the overhead in messages per second per node. The results show that fine-grained topology adaptation performs better than using super-peers or not exploiting heterogeneity. HeteroPastry achieves significantly higher success rate, and lower delay and overhead than SuperPastry and Pastry. They also show that search in structured graphs can perform better than in unstructured graphs.

HeteroPastry achieves the highest success rate, the lowest delay, and the lowest overhead. This demonstrates that HeteroPastry can exploit heterogeneity effectively to improve scalability; the high success rate indicates that the bound on the length of random walks can be small and the low delay shows that they are likely to terminate early, which results in low overhead. The other systems would require longer random walks to achieve the success rate of HeteroPastry, which would increase their overhead.

All the graph maintenance algorithms benefit from suppression of failure detection traffic by query traffic. For example, Gia's overhead without queries is approximately twice the overhead of Gnutella 0.6. The overheads of the two are comparable with queries because of the suppression of failure detection traffic and shorter random walks. HeteroPastry

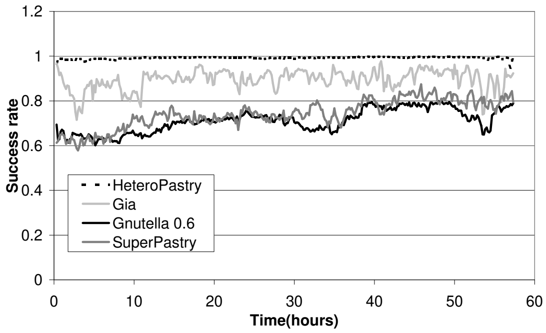

Figure 11: Query success rate.

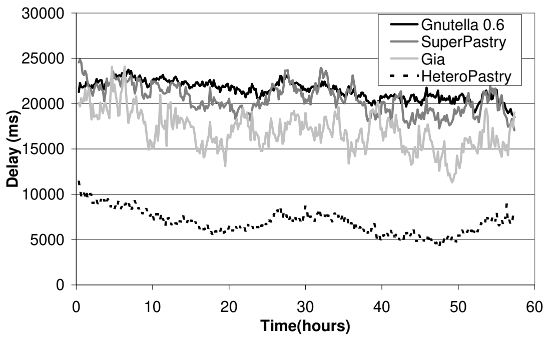

Figure 12: Query delay for successful queries.

has lower overhead than Pastry (compare Figure 8 with Figure 13) because random walks are shorter as demonstrated by the lower delay.

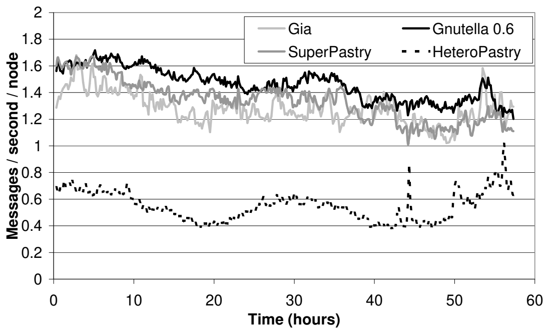

Figure 13: Messages per second per node.

So far we have considered the overhead averaged over all live nodes in each 10 minute window in the trace. Since both Gia and HeteroPastry adapt the topology to distribute load according to node capacity, we looked at the distribution of the number of messages per second per node in the ten minutes preceding the 5 hour mark in the trace. The total number of messages received in this 10 minute window was 2.4 times higher for Gia than HeteroPastry. Figures 14 and 15 show the cumulative distribution of the number of messages per second per node for each capacity value in HeteroPastry and Gia.

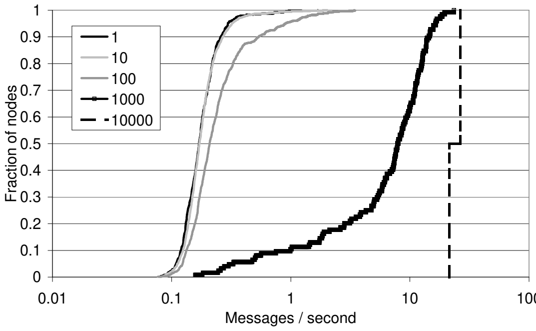

Figure 14: Cumulative distribution of messages per second per node for each capacity value in HeteroPastry.

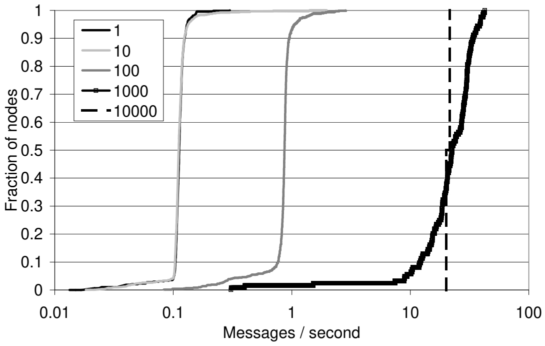

Figure 15: Cumulative distribution of messages per second per node for each capacity value in Gia.

The maximum message rate observed was only 42.63 for Gia and 26.48 for HeteroPastry. Both systems do a good job of distributing message load according to capacity; nodes with higher capacity receive more messages. The message rate for nodes with capacity 1 is low; the median is only 0.17 and the 95th percentile is only 0.30 in HeteroPastry, and the median is 0.11 and the 95th percentile is 0.13 in Gia. For the nodes with capacity 10 in HeteroPastry, the median is also 0.17 and the 95th percentile is 0.32, and the median is 0.11 and the 9th percentile is 0.14 in Gia. Since the indegree of 1- and 10-capacity nodes is bounded to the same value, this is not surprising. In both Gia and HeteroPastry, the 100-capacity nodes incur a higher overhead than the 1- and 10-capacity nodes but a lower overhead than the 1000-capacity nodes.

The figures also show that the load on any node is sufficiently low (with a query rate of 0.01 queries per second per node) that flow control is not necessary. Gia's flow control mechanism [10] can be applied to HeteroPastry to enable scaling to higher query rates.

We also studied the distribution of replicas of node indices, which is another indicator of the effectiveness of both systems in adapting the topology to diffferent node capacities.

	Capacity	1	10	100	1000	10000
Gia	Mean	3	3	23.56	126.02	128
	Median	3	3	24	128	128
	95th	3	3	25	128	128
Hetero-Pastry	Mean	2.15	2.38	14.50	104.66	128
	Median	2	3	15	25	128
	95th	3	3	24	128	128

Table 2: Distribution of replicas of node indices for different capacity values in Gia and HeteroPastry.

Table 2 summarises the distribution of replicas of indices for each capacity value in both systems. The total numbers of index replicas is 27,707 in HeteroPastry and 38,153 in Gia. Both systems do a good job at distributing index replicas (and indegree) according to node capacity. Gia replicates more because it is more effective at pushing replicas to nodes with capacity 100 and 1000.

Poisson traces The experiments described so far use a trace of node arrivals and departures collected in a real Gnutella deployment. The next set of experiments compare the performance of Gia and HeteroPastry using artificial traces with more nodes and different rates of churn. These traces have Poisson node arrivals and an exponential distribution of node session times with the same rate. We generated traces with session times of 5, 15, 30, 60, 120 and 600 minutes and in all cases the average number of nodes was 10,000. We used the same data and query distribution as in the previous experiments. It is important to note that a session time of 5 minutes is very small; it is 28 times smaller than the average session time of 2.3 hours in the Gnutella trace.

Figure 16 shows the total number of messages per second per node for the different session times. Both Gia and HeteroPastry have low overhead across all session times.

Gia's overhead is almost constant across all session times. Short session times increase Gia's overhead because of increased retransmissions and traffic to fill neighbour tables. However, this is offset by a decrease in fault detection traffic due to a decrease in the average number of neighbours; there are 15.1 neighbours when the session time is 600 and 10.7 when it is 5.

HeteroPastry has a lower message overhead than Gia for session times of 30 minutes or greater. This overhead decreases between 60 and 600 minutes because HeteroPastry adapts the routing table probing rate to match the failure rate. HeteroPastry incurs a higher message overhead than Gia for extremely high churn rates mostly due to the overhead of maintaining the leaf set. This overhead could be reduced without impacting query success rate and delay by using a smaller leaf set or disabling the mechanisms to ensure strong leaf set consistency [5], which are not important in this application.

Figure 17 shows the lookup success rate for the different ses-

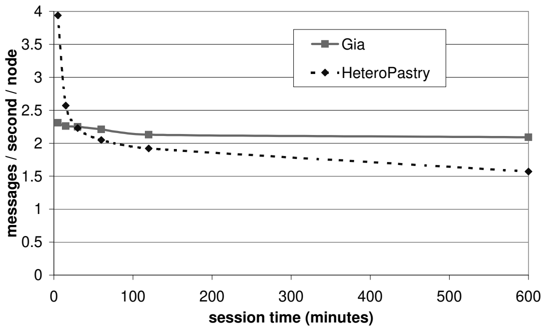

Figure 16: Messages per second per node for Gia and HeteroPastry versus session time.

sion times. As in previous experiments, HeteroPastry achieves a success rate higher than Gia across all session times.

The success rates with 10,000 nodes are lower than those observed before because there are more nodes. There are only between 1,300 and 2,700 active nodes at any time in the Gnutella trace. This also results in higher message overhead with 10,000 nodes even with a session time of 600 minutes.

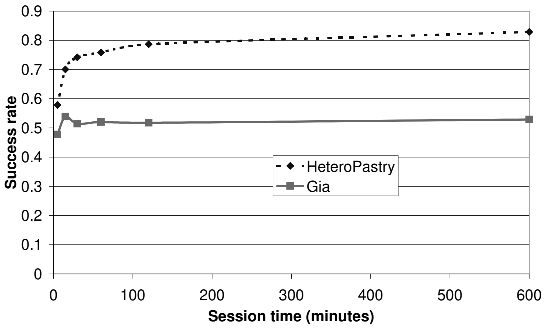

Figure 17: Query success rate for Gia and HeteroPastry versus session time.

The delay incurred for successful lookups is similar in both HeteroPastry and Gia. HeteroPastry achieves a lower average delay per lookup because it has a higher success rate and failed lookups take longer to complete on average than successful lookups. A failed lookup will take at least 64,000ms to complete on average because the random walk has 128 hops and the average delay per hop is 500ms. Message losses and failures can increase this value further. Therefore, HeteroPastry achieves a delay at least 12% lower than Gia with 5 minute session times and at least 43% lower with 600 minutes session time.

4.3.3 Constrained floods

We also compared the performance of constrained flooding and random walks in HeteroPastry. We configured constrained floods to visit at most 128 nodes as with the random walks. Both algorithms visit exactly the same 128 nodes when the query fails so they have the same success rate.

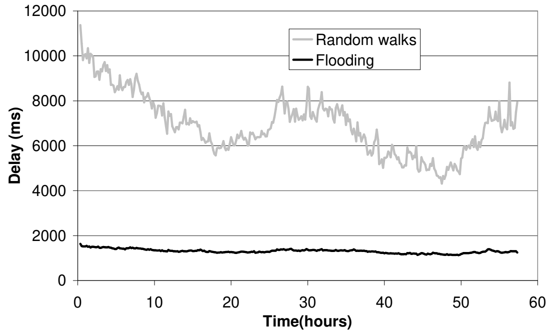

Figure 18: Query delay when using constrained flooding and random walks in HeteroPastry.

Figure 18 shows the delay for successful queries using both constrained floods and random walks. It shows that constrained flooding can locate content faster than random walks. This is not surprising because constrained flooding visits nodes in parallel; all 128 nodes are visited after only 7 hops. It takes 128 hops to visit all the nodes with the random walk. Additionally, random walks use acknowledgments and retransmissions to recover when the query is forwarded to a node that fails. This introduces delays that increase when the failure rate in the trace increases (as shown in Figure 18). The delay of constrained floods remains constant because we do not use acknowledgments and retransmissions and instead rely on redundancy to cope with node failures. We observed the same success rate for both flooding and random walks, which demonstrates the effectiveness of using redundancy to cope with node failure during constrained floods.

Figure 19 shows the number of messages per second per node when using constrained floods and random walks in HeteroPastry. It demonstrates the advantage of random walks over flooding; random walks result in lower overhead because they stop when they find a copy of the file and visit less nodes than constrained floods on average. It is interesting to note that the overhead with constrained floods is comparable to the overhead in the unstructured overlays. Additionally, some peer-to-peer applications discover multiple nodes with matching content, for example, to enable more efficient downloads with a some form of striping. The benefit of random walks over constrained floods decreases in this case. Constrained floods are likely to be the best strategy for many applications.

5 Future work

It would be interesting to perform a comparative study of the security properties of structured and unstructured graphs. Previous work [30, 6] identified several attacks and proposed solutions to secure structured overlays. The solutions proposed in [11, 6] can be used to protect both structured and unstructured graphs from Sybil attacks [11] and removing constraints

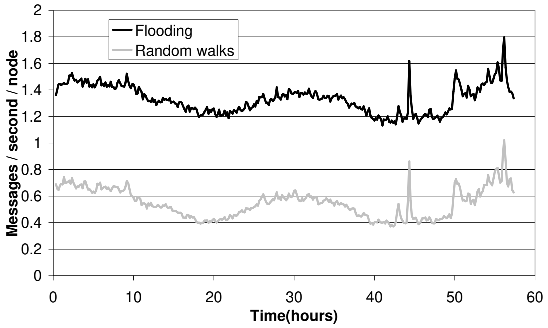

Figure 19: Messages per second per node when using constrained floods and random walks in HeteroPastry.

on data placement removes several attacks.

However, there is an attack that seems hard for unstructured graph maintenance algorithms to defend against. These algorithms populate their neighbour tables by obtaining neighbours from (or through) other nodes. The problem is that some of these nodes may be malicious. In an overlay where a fraction f of the nodes is malicious, honest nodes return malicious nodes with probability f but malicious nodes can return malicious nodes with probability 1. The result is an increasing fraction of malicious neighbours in the neighbour tables of honest nodes. There are mechanisms to protect structured overlays against this attack in [6] but they rely on a constrained routing table that imposes very strong structural constraints on the graph. This defence cannot be applied to unstructured overlays and it disallows the topology adaptation technique that we used in HeteroPastry. Another problem with topology adaptation is that malicious nodes can advertise large capacities to be inserted in the neighbour tables of honest nodes and disrupt data discovery. Security is not a large concern in current applications but these problems must be addressed before several interesting applications can be supported.

HeteroPastry provides efficient support for complex queries but it also provides applications with the option to leverage the efficient exact-match queries supported by the structured graph. For example, applications could leverage exact-match queries to find rare files while using constrained floods and random walks to find popular ones. This is an interesting area of future work.

6 Conclusion

It is commonly believed that unstructured graphs cope with churn better, exploit heterogeneity more effectively, and support complex queries more efficiently than structured graphs. This paper shows that this is not a fundamental problem.

This paper presented a detailed simulation driven comparison of structured and unstructured graphs. The simulation used a real-world trace and showed that structured graphs can cope with churn better than unstructured graphs. Then, a structured graph maintenance protocol was extended to exploit heterogeneity to improve scalability and validated its performance and effectiveness under churn. Finally, we developed a hybrid system that uses a structured graph but exploits the data placement and search mechanisms currently used with unstructured overlays. Simulation results using a real-world trace show that the hybrid system can support complex queries with lower message overhead while providing higher query success rates and lower response times than systems based on unstructured graphs.

References

[1] The Gnutella 0.4 protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[2] The Gnutella 0.6 protocol specification, 2002. http://www.limewire.org/.

[3] Kazaa, 2002. http://www.kazaa.com/.

[4] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *IPTPS'03*, Feburary 2003.

[5] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft Research, Dec. 2003.

[6] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, Dec. 2002.

[7] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, Aug. 2003.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[9] 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'03*, Apr. 2003.

[10] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like p2p systems scalable. In *Proc. ACM SIGCOMM*, Aug. 2003.

[11] J. R. Douceur. The Sybil attack. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, Mar. 2002.

[12] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient broadcast in structured p2p networks. In *IPTPS'03*, Feb. 2003.

[13] F. L. Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulie. Clustering in peer-to-peer file sharing workloads. In *IPTPS'04*, Feburary 2004.

[14] P. Ganesan, Q. Sun, and H. Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *Infocom'03*, Apr. 2003.

[15] O. Gnawali. A keyword set search system for peer-to-peer networks, 2002. Master Thesis, MIT.

[16] The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[17] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *ACM SIGCOMM 2003*, 2003.

[18] P. K. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. SOSP 2003*, Lake Bolton, NY, USA, Oct. 2003.

[19] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *IPTPS'02*, Mar. 2002.

[20] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS'03*, Feb. 2003.

[21] Q. LV, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. 16th ACM International Conference on Supercomputing (ICS'02)*, June 2002.

[22] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make Gnutella scalable? In *Proc. IPTPS'02*, Cambridge, MA, USA, Feb. 2002.

[23] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, Feb. 2003.

[24] 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.

[25] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *NGC*, Nov. 2001.

[26] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Middleware 2003*, 2003.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[28] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, Jan. 2002.

[29] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Internet Measurement Workshop*, 2002.

[30] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, Mar. 2002.

[31] 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.

[32] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *ACM SIGCOMM 2003*, Aug. 2003.

[33] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, San Francisco, California, 1996.

[34] 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, Apr. 2001.