Should we build Gnutella on a structured overlay?

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

Abstract

There has been much interest in both unstructured and structured overlays recently. Unstructured overlays, like Gnutella, build a random graph and use flooding or random walks on the graph to discover data stored by overlay nodes. Structured overlays assign keys to data items and build a graph that maps each key to a specific node. The structure of the graph enables efficient discovery of data items given their keys but it does not support complex queries.

Should we build Gnutella on a structured overlay? We believe the answer is yes. We replaced the random graph in Gnutella by a structured overlay while retaining the content placement and discovery mechanisms of unstructured overlays to support complex queries. Our preliminary results indicate that we can use structure to improve the performance of floods and random walks. They also indicate that structure can be used to reduce maintenance overhead, which is surprising because it is commonly believed that unstructured overlays have lower maintenance overhead than structured overlays.

1 Introduction

In recent years, there has been much interest in peer-to-peer (p2p) overlays because they provide a good substrate for building large scale data sharing and content distribution applications. There are two types of overlays: unstructured and structured.

Unstructured overlays, like Gnutella [1], are widely used and there has been a large amount of work on improving Gnutella, for example, [6, 8, 9]. Unstructured overlays organize nodes in a random graph and use flooding or random walks on the graph to query content stored by overlay nodes. Each visited node evaluates the query locally on its own content. This supports arbitrarily complex queries but it is inefficient because queries for content that is not widely replicated must be sent to a large fraction of nodes.

Structured overlays [11, 15, 13, 17] were developed to overcome the performance inefficiencies of unstructured overlays. They assign keys to data items and organize the overlay nodes into a graph that maps each key to a responsible node. The graph is structured to enable efficient discovery of data items given their keys but it does not support complex queries. Additionally, it is necessary to store a copy or a pointer to each data item at the node responsible for the item's key.

This paper argues that we should build Gnutella on a structured overlay. We replace the random graph in Gnutella by a structured overlay while retaining the content placement and discovery mechanisms of unstructured overlays to support complex queries. We call the new system Structella.

Like Gnutella, Structella does not use structure to organize the content in the overlay. Each node stores its own content and it does not store copies or pointers to it on other nodes. Structella also uses either a form of flooding or random walks to discover content 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. We provide preliminary results comparing the query overheads of Structella and an optimized version of Gnutella. The results indicate that there is a significant performance improvement when using our system.

It is commonly believed that unstructured overlays have lower maintenance overhead than structured overlays, especially when there is a high churn rate. This could negate the performance benefit that we observed for queries in Structella. We show how to use structure to reduce maintenance overhead in structured overlays. We compare the maintenance overheads of Structella and the optimized version of Gnutella using a real world trace of node arrivals and departures in a Gnutella overlay. The results show that Structella incurs lower overhead. So, we see no reason to build Gnutella on top of an unstructured overlay!

In Section 2, we describe the structured overlay that we use to build Structella and the unstructured overlay that we use for comparison. Section 3 compares the maintenance algorithms and overheads of both systems, and Section 4 discusses how we implement complex queries in Structella and evaluates their performance. Finally, we conclude and describe future work in Section 5.

2 Background: Overlays

We used Pastry to implement Structella but we could have used most of the other structured overlays. We chose Pastry because it has low maintenance overhead [10]. This section provides a brief description of Pastry and the optimized version of Gnutella that we used to evaluate Structella.

2.1 Pastry

Pastry uses a circular 128-bit id space. It assigns each overlay node a nodeId that is chosen randomly with uniform probability from the id space. Keys are chosen from the same space. NodeIds and keys are interpreted as a sequence of digits in base 2^b. We use $b = 1$ in the experiments in this paper. Given a message and a destination key, Pastry routes the message to the node whose nodeId is numerically closest to the key.

Each node maintains a routing table and a leaf set to route messages. 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 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. We implemented Structella using a version of Pastry that does not use proximity-aware routing. The routing table entries are filled with a random node with the required prefix match.

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 $l = 8$ in the experiments in this paper.

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. A full description of Pastry can be found in [13, 3].

2.2 Gnutella

In order to evaluate the performance and overhead of Structella, we implemented a Gnutella-like unstructured overlay. The implementation is based on the Gnutella 0.4 protocol [1], except that it was optimised to improve query performance and reduce maintenance overhead.

Gnutella is based on 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. The neighbour tables are designed to be symmetric in order to reduce maintenance load.

There is an upper and lower bound on the number of entries in each node's neighbour table. We use a lower bound of 4 and an upper bound of 8. These bounds were chosen based on the values used in other unstructured overlays.

3 Overlay maintenance

It is necessary to maintain the overlay to ensure that it remains working as nodes join and leave. This section compares the mechanisms used by the Gnutella-like system and by Pastry (and Structella) to maintain the overlay.

To join either overlay, the joining node contacts a *bootstrap* node that is randomly chosen from the current members of the overlay. In the Gnutella-like system, the bootstrap node initiates a search for overlay nodes that have less neighbour table entries than the upper bound. The original Gnutella uses flooding to discover these nodes. We evaluated two different versions: flooding and random walk.

In the flooding version, the bootstrap node floods a message with a time-to-live (TTL) to all nodes in its neighbour table. Each node that receives the message decrements the TTL. If the result is greater than zero and the message has not been received by this node before, it sends the message to all entries in its neighbour table except to the sender. The nodes that receive the message also check if the number of entries in their neighbour table is below the upper bound and if it is, they send a reply message back to the joining node. The joining node inserts the nodes that reply in its neighbour table until the number of entries reaches the upper bound. It is likely to receive replies from nodes that it does not insert in the neighbour table. These nodes are stored in a cache and are used to replace failed neighbours. We used TTL=4 in the experiments described in this paper.

In the random walk version, the bootstrap node chooses one of the entries in its neighbour table at random and sends the message to that node. The message includes a counter that is initialized to the number of neighbours being sought by the joining node. Each node that receives this message checks if the number of entries in its neighbour table is below the upper bound. If it is, the node sends a reply message back to the joining node and decrements the message counter. If the message counter drops to zero, the message is not forwarded further; otherwise, it is forwarded to a randomly selected node in the neighbour table. As with flooding, the joining node adds the nodes it receives replies from to its neighbour table.

Both approaches have the side effect that neighbour table entries are likely to point to nodes that are close to the bootstrap node. This can increase the probability of partitioning the overlay when there are node failures but it should not affect content discovery. This could be avoided by refining the joining approaches at the expense of increased overhead. Since we do not evaluate the resilience of the overlay, we opted for the less expensive approaches described.

Node joining in Pastry exploits the overlay structure. 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. x obtains the ith row of its routing table from the ith node encountered along the route from a to z and it obtains the leaf set from z.

In addition to node joining, overlays need to detect failures and repair faulty neighbours. The simplest approach to detect failures is to periodically send a probe request to each neighbour and wait for a reply. We exploit the symmetry of neighbour tables in the Gnutella-like overlay to reduce failure detection overhead. Periodically, each node sends an *I'm alive* message to every node in its neighbour table. Since its neighbours do the same, each node should receive a message from each neighbour in each period. If it does not receive this message, it probes the node and if the node does not reply it marks it faulty. In the experiments in this paper we set the period to 30 seconds. Faults are repaired using cached nodes when available or using a flood or random walk from a known node (depending on the version).

Pastry uses different strategies to detect failures in the routing table and the leaf set. Since the routing table is not symmetrical, a node explicitly probes every member every t_r seconds to detect failures. But we exploit the structure of the overlay to detect failures in leaf set members. 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 and informs all the members of the new leaf set about the failed node. In the experiments $t_l = 30$ seconds and t_r is set dynamically by each node based on the node failure rate in the overlay observed by the node [10]. We configured Pastry to achieve a 10% loss rate.

In both the Gnutella-like implementation and Pastry, messages sent between the nodes are used to replace explicit fault detection messages.

3.1 Maintenance overhead: preliminary results

We ran an experiment to evaluate the maintenance overhead of both the Gnutella-like overlay and Structella (Pastry). We used a packet-level discrete-event simulator with a transit-stub network topology model [16]. This model has 5050 routers arranged hierarchically. 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 [16]. The simulator models the propagation delay on the physical links. The average delay of core 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.

We obtained a trace of node arrivals and failures from a measurement study of Gnutella. The study [14] 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 1300 and 2700. The failure rate and arrival rates are similar but there are large daily variations (more than a factor of 3). We used this trace to control node arrivals and failures in our experiment for both systems. There was no query traffic during this experiment.

Figure 1 shows the average number of messages per second per node for Structella and for the Gnutella-like unstructured overlay using both flooding and random walks to build the neighbour table. The x-axis represents simulation time. In all three systems, most of the overhead is due to fault detection messages. The results demonstrate the benefit of exploiting overlay structure to reduce maintenance overhead.

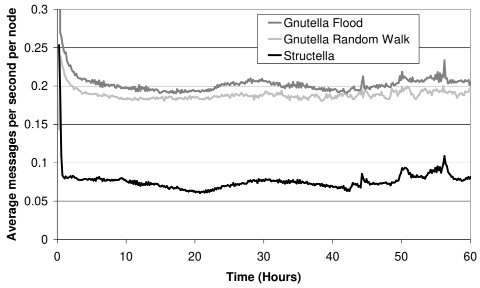

Figure 1: Maintenance overhead in messages per second per node over time.

4 Content Discovery

Structella is implemented on top of Pastry and inherits its low maintenance overhead. It uses Pastry's overlay routing and maintenance algorithms but it does not use structure to organize the content in the overlay. Each node stores its own content and it does not store copies or pointers to it on other nodes when it joins. This reduces maintenance overhead and ensures that the content is distributed across the nodes in the overlay independently from their nodeIds.

We could implement support for structured queries using exactly the same flooding and random walk techniques used in Gnutella, but can we exploit the structure in the overlay to do better?

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; the time-to-live field provides only very coarse control.

We can do better by replacing flooding with the broadcast mechanisms that have been proposed for structured overlays [12, 5, 7]. Structella uses Pastry's broadcast mechanism [5] to broadcast queries to overlay nodes. The nodes that receive the query evaluate it against the local content and send matching content back to the sender. This broadcast is efficient because it exploits the overlay structure; it sends approximately N messages in about $logN$ hops and it does not require any state beyond the routing state already maintained by Pastry.

The broadcast works as follows. A node y broadcasts a message by sending the message to all the nodes x in its routing table. Each message is tagged with the routing table row r of node x. When a node receives a message tagged with r, it forwards the message to all nodes in its routing table in rows greater than r. This continues until a node receives a message tagged with r and it has no entries in rows greater than r. Usually, each node receives the message only once but the technique to deal with empty routing table slots [5] may result in a small number of duplicate messages.

The other technique used for searching unstructured overlays like Gnutella is the random walk. We implement random walks in Structella 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. When a node receives a query in a random walk, it uses the leaf set to forward the query to its left neighbour in the id space. It also evaluates the query against the local content and sends matching content back to the query originator. We terminate the random walk when we find matching content. This approach ensures that each node is visited only once and it can be trivially extended to use multiple concurrent random walks to improve query times while ensuring that each random walk explores a disjoint set of nodes. Concurrent or long random walks in unstructured overlays have a non-negligible probability of visiting the same node more than once.

One of the advantages of random walks over flooding in unstructured overlays is that random walks provide more precise control over the number of overlay nodes that is visited to satisfy a query. It is possible to constrain floods using the TTL mechanism but this provides only very coarse-grained control. By exploiting structure, we can implement a flooding algorithm that provides very accurate control over the number of visited nodes. For example, we can use the broadcast mechanism described above with an upper bound on the row number of entries to which the query is forwarded. This enables the query originator to choose the number of nodes visited from the set of powers of 2^b. This can be extended to provide arbitrarily fine grained control over the number of nodes visited. We refer to this as a *constrained flood*.

4.1 Query performance: preliminary results

We ran experiments to compare query performance in Structella and the Gnutella-like overlay. We started by creating 20,000 nodes and then we ran the query experiments on a stable overlay without joins or leaves. Otherwise, the experimental setup was similar to the one described above. The average number of links per node in the Gnutella-like overlay was 7.8.

The performance of queries depends on the number of nodes that store content matching the query. We ran queries with 0, 1, 200, 2000, and 5000 nodes with content matching the query (0%, 0.005%, 1%, 10% and 25% of the nodes in the overlay). The results that we present are the average for 100 queries. The queries originated from a node selected randomly from the overlay nodes without matching content.

We evaluate both flooding and random walks in the Gnutella-like overlay, and we evaluate flooding, constrained flooding, and random walks in Structella. We bound the number of nodes to visit when using random walk or constrained flooding to 128. The Gnutella-like overlay uses query id caching to suppress duplicates; if a node has received a query message, it ignores all further copies it receives. In both overlays, a node stops forwarding a query if it has matching content.

Figure 2 shows the average number of messages required when flooding a query with a varying fraction of nodes with matching content. In all cases (except 0%), all queries suc-

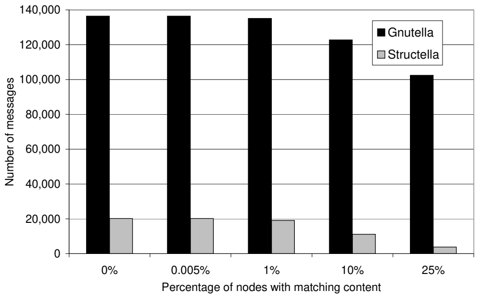

Figure 2: Number of messages per query with flooding.

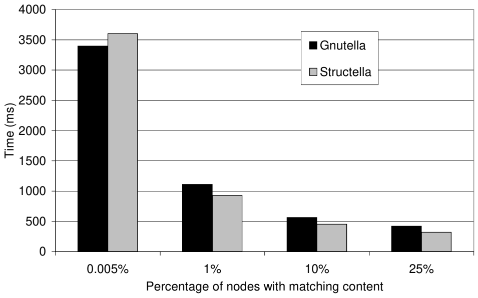

Figure 3: Query delay with flooding.

ceeded in finding matching content.

When no nodes have matching content, the number of messages in the Gnutella-like overlay is 136,456. This is expected because a node forwards a query message to all nodes in its neighbour table (except the node from which it received the query) and it discards any duplicates it receives. Since nodes have on average 7.8 neighbours, the expected number of messages is $(7.8 - 1) \times 20,000$. For Structella, the number of messages is only 20,125 where the 125 extra messages are due to missing entries in routing tables.

The results also show that the query overheads remain high when only a few nodes have content matching the query. Even when 25% of the nodes have matching content, the number of messages in the Gnutella-like overlay is 102,391. This is also expected because 25% of the nodes have matching content and do not forward the query message. Since nodes have on average 7.8 neighbours, the expected number of messages is $(7.8-1) \times 20,000 \times 0.75$. When 10% of the nodes match the content, Structella flooding uses an order of magnitude less messages and over 26 times less messages when 25% of the nodes match the content. We therefore conclude that exploiting structure when flooding provides a large reduction on query overhead.

Figure 3 shows delay between the time a query is issued and the time when the first node with matching content receives the query. The delays are similar as expected.

Figure 4 shows the average number of messages required per query when using the random walk and constrained flooding. The success probability of the queries was very similar for the three techniques. As expected, the number of messages used by the random walks is similar in Structella and the Gnutella-like overlay. However, the message overhead is higher with constrained flooding because query forwarding does not stop when the first node with matching content is found.

The benefit of constrained flooding can be seen in Figure 5, which shows delay between the time a query is issued and the time when the first node with matching content receives the query. The random walks in Structella and the Gnutella-like overlay perform similarly. However, constrained flooding in Structella performs significantly better, especially when there are few nodes with matching content. The improvement is due to parallelism in the search.

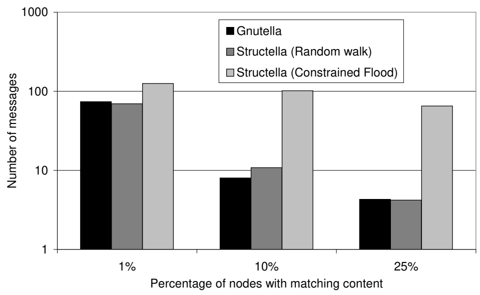

Figure 4: Number of messages per query with random walk or constrained flooding.

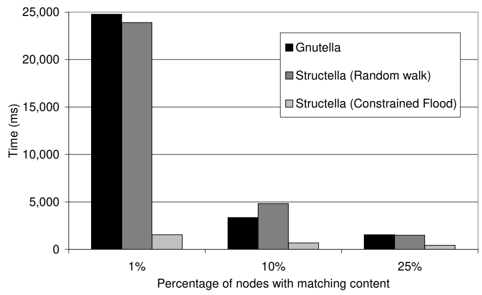

Figure 5: Query delay with random walk or constrained flooding.

Whilst the results are preliminary, they suggest that exploiting structure improves query performance and reduces maintenance overhead.

5 Conclusions and Future Work

This paper argued that we should build Gnutella on a structured overlay. There are two perceived obstacles to doing this: *(i)* the high maintenance overhead of structured overlays; and *(ii)* the lack of support for complex queries in structured overlays. Structella addresses these issues. It replaces the random graph in Gnutella by a structured overlay but it does not use the structure to organize the content. Structella supports complex queries using variants of flooding and random walks like Gnutella 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. Structella also leverages the structured overlay to reduce the maintenance overheads. This results in significant performance improvements for complex queries and lower maintenance overhead than Gnutella.

The results and ideas presented in this paper are preliminary. A number of issues require further investigation. It is important to study the behavior of Structella's search mechanisms with frequent node arrivals and departures. We also plan to improve our Gnutella-like overlay by adopting the techniques proposed in [6] and version 0.6 of the Gnutella protocol. In particular, Gnutella 0.6 introduces the concept of ultra-peers, which are high capacity nodes that act as proxies for lower capacity nodes. There are several ways to extend Structella to support ultra-peers; the simplest is to form a structured overlay containing only ultra-peers and attach other peers to ultra-peers as done in Gnutella 0.6. We could also exploit efficient anycast implementations in structured overlays [4] to help balance load across ultra-peers. Another interesting issue to explore is whether structure can help protect against malicious nodes in the overlay. There have been proposals on how to secure structured overlays in the presence of malicious nodes [2], which we could potentially exploit. Finally, the current version of Structella uses a variant of Pastry that does not exploit network locality. Future work will examine the benefit and cost trade-offs of exploiting locality in Structella.

Should we build Gnutella on a structured overlay? So far, we see no reason to build Gnutella on top of an unstructured overlay!

References

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

[2] 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)*, 2002.

[3] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, May 2002.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application level anycast for highly dynamic groups. In *NGC'2003*, 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'03*, Apr. 2003.

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

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

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

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

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

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.