Should we build Gnutella on a structured overlay?

Ant Rowstron

joint work with
Miguel Castro, Manuel Costa
Microsoft Research Cambridge
Structured P2P overlay networks

- Structured overlay network maps keys to nodes
- Routes messages to keys; (can implement hash table)

[CAN, Chord, Kademlia, Pastry, Skipnets, Tapestry, Viceroy]
Mapping keys to nodes

- Large **id space** (128-bit integers)
- **NodeIds** picked randomly from space
- Key is managed by its **root node**:
  - Live node with id closest to the key
Pastry

<table>
<thead>
<tr>
<th></th>
<th>0*</th>
<th>1*</th>
<th>2*</th>
<th>3*</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20*</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>2030</td>
<td>2031</td>
<td>2032</td>
<td>2033</td>
<td></td>
</tr>
</tbody>
</table>

- **routing table**
  - nodeIds and keys in some base $2^b$ (e.g., 4)
  - prefix constraints on nodeIds for each slot

203231
nodeId

leaf set
Structured overlays

• Overlay topology
  – nodes self organize into structured graph
  – node identity constrains set of neighbors

• Data placement
  – data identified by a key
  – data stored at node responsible for key

• Queries
  – efficient key lookups (O(logN))

examples: CAN, Chord, Pastry, Tapestry
Gnutella

- Nodes form random graph (unstructured overlay)
- Node stores its own published content
- Lookups flooded through network (inefficient)
Gnutella

- Nodes form random graph (unstructured overlay)
- Node stores its own published content
- Lookup using random walks (needles and haystacks!)
Unstructured overlay

• Overlay topology
  – nodes self-organize into random graph
• Data placement
  – node stores data it publishes
• Queries
  – overlay supports arbitrarily complex queries
  – floods or random walks disseminate query
  – each node evaluates query locally

example: Gnutella
Can we build Gnutella on a structured overlay?

• Complex queries are important
  – unstructured overlays support them
  – structured overlays do support them

• Peers are extremely transient
  – unstructured overlays more robust to churn
  – structured overlays have higher overhead

[Chawathe et al. SIGCOMM’03]
Complex queries

• Arbitrarily complex queries
  – Unstructured overlay
    • Flood
      – High overhead due to duplicates
    • Random walks
      – High lookup latency
  – Support arbitrarily complex queries
  – Structured overlays
    • ?
Complex queries (structured)

• **Structured overlay topology**
  – nodes self organize into structured graph

• **Same data placement as unstructured**
  – node stores data it publishes

• **Same queries as unstructured**
  – overlay supports arbitrarily complex queries
  – floods or random walks disseminate queries
  – each node evaluates query locally
Flood queries

- Exploit structure to avoid duplicates
Flood queries
Random walk queries 1
Random walk queries 2
Random walk queries 3

- Exploiting routing tables
- Breadth-first search
Story so far….

- Gnutella is built using an unstructured overlay
- Described hybrid approach
  - Structured overlay graph
  - Unstructured overlay data placement
- Described how to exploit structure in lookup
  - Same techniques as in an unstructured overlay
  - Implemented more efficiently

Next part: Churn and overhead
Overhead

• Both structured and unstructured
  – detect failures
  – repair overlay graph when nodes join or leave
Detecting failures

- Probe neighbors in overlay

- Exploit symmetric state
  - Heartbeats versus probes

- Number of heartbeats is number of neighbors

- Suppress heartbeats with application traffic
Exploiting structure for maintenance

- Heartbeat sent to neighbor on the left
- Probe node if no heartbeat
- Tell others about failure if no probe reply

- Leads to lower overhead
Comparing overhead

- **Unstructured overlay (Gnutella 0.4)**
  - Max and min bounds placed on # neighbors
  - Node discovery on join using random walks
  - Failure detection heartbeat every 30 seconds

- **Structured overlay (MS Pastry)**
  - Leafsets
    - Failure detection using heartbeats every 30 seconds
  - Routing table
    - Failure detection using probes (tuned to churn)
Experimental comparison

• Discrete event simulator
  – Transit-stub network topology

• UW trace of node arrivals and departures
  – [Saroiu et al. MMCN’02]
  – 60 hours trace
  – average session = 2.3 hours, median ~ 1 hour
  – Active nodes varies between 2,700 and 1,300
Gnutella trace: Failure rate

Node failures per second per node

Time (Hours)
Overhead: Configuration

- Gnutella 0.4 (4)
  - Min neighbors 4, max neighbors 8 (avg. 5.8)
- Gnutella 0.4 (8)
  - Min neighbors 8, max neighbors 32 (avg. 11)
- Pastry
  - $b=1$, no proximity neighbor selection, $l = 32$
Overhead: Maintenance

![Graph showing messages per second per node over time for Gnutella 0.4 (8) and Gnutella 0.4 (4) compared to Pastry.]

- Gnutella 0.4 (8)
- Gnutella 0.4 (4)
- Pastry
Gnutella 0.6 (SuperPeers)

• Super peers form random graph
  – Uses Gnutella 0.4 algorithm

• Normal nodes use super peers as proxies
  – Failure detections using heartbeats (30 secs)
  – Connect to multiple super peers
SuperPastry

- Super peers form Pastry overlay
- Normal nodes use super peers as proxies
  - Failure detections using heartbeats (30 secs)
Overhead: Configuration

• 0.2 probability of node being a super peer
• Gnutella 0.6 configured:
  – Min neighbours = 10
  – Max neighbours = 32
• SuperPastry configured
  – Max in-degree from routing table = 32
• Super peers proxy for 30 normal nodes
• Normal nodes pick 3 super peers
Overhead: Maintenance

![Graph showing messages per second per node over time for Gnutella 0.6 and SuperPastry. The graph illustrates the maintenance overhead for both systems.](image-url)
Gia [Chawathe et al. SIGCOMM’03]

- Adapts overlay to exploit heterogeneity
  - Uses a per-node metric of satisfaction
  - Seeks new neighbors if unsatisfied
  - Use parameters in Sigcomm Paper
  - Neighbors [min = 3, max = max(3,min(128,C/4)) ]
    - Average 15.8

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Probability</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.20</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>0.45</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>0.30</td>
<td>25</td>
</tr>
<tr>
<td>1000</td>
<td>0.049</td>
<td>125</td>
</tr>
<tr>
<td>10000</td>
<td>0.001</td>
<td>128</td>
</tr>
</tbody>
</table>
HeteroPastry

- Routing table neighbor selection using capacity metric
- Uses routing table in-degree bound
  - Calculated as for Gia
Overhead: Maintenance

![Graph showing overhead maintenance for Gia and HeteroPastry. The x-axis represents time in hours, ranging from 0 to 60, and the y-axis represents messages per second per node, ranging from 0 to 0.8. The graph compares the overhead of Gia (red line) and HeteroPastry (black line).]
The story so far....

- Both structured and unstructured
  - detect failures
  - repair overlay graph when nodes join or leave
- Structured exploits structure
  - Lower overheads
- Unstructured overlays sensitive to neighbors choice
  - Random walks between node discovery

Finally: Putting it all together....
Search: Configuration

- eDonkey file trace [Fessant et al. IPTPS’04]
  - 37,000 peers (25,172 contribute no files)
  - 923,000 unique files (heavy tail zipf-like)
- Each node performs 0.01 lookups per second (using a Poisson process)
  - Random walks TTL 128
- One hop replication [Chawathe et al. SIGCOMM’03]
  - Uses routing table in structured overlays (***)
Search: Messages

![Graph showing search messages over time for different systems: Gia, Gnutella 0.6, SuperPastry, and HeteroPastry. The graph plots messages per second per node against time in hours.]

- **Gia**
- **Gnutella 0.6**
- **SuperPastry**
- **HeteroPastry**
Search: Success rate

![Graph showing search success rate over time for HeteroPastry, Gia, Gnutella 0.6, and SuperPastry. The graph plots success rate against time (in hours) with different line colors representing each technology. HeteroPastry maintains a high success rate throughout the time period, while Gia and SuperPastry show fluctuations with lower success rates. Gnutella 0.6 is somewhere in between.](image-url)
Search: Delay
Conclusions

• Structure can improve Gnutella
  – Handles transient peers well
  – Exploits structure to reduce maintenance overhead
  – Supports complex queries
  – Can also support DHT functionality
  – Can exploit heterogenity
And finally a question…

Does structure make security easier?

For slides:

http://www.research.microsoft.com/~antr/camb-ast.ppt

For more information:

http://www.research.microsoft.com/~antr/Pastry
Flooding queries

- exploit structure to avoid duplicates

- flooding a query $q$
  - if node is source of $q$ do
    for each routing table row $r$
    send $<\text{flood}, q, r>$ to nodes in row $r$
  - if node receives $<\text{flood}, q, s>$ do
    for each routing table row $r$ such that $r > s$
    send $<\text{flood}, q, r>$ to nodes in row $r$

- recursively partitions nodes into disjoint sets