Acknowledgments: Lecture slides are from the graduate level Computer Networks course taught by Srinivasan Seshan at CMU. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.
Overview

• P2P Lookup Overview
• Centralized/Flooded Lookups
• Routed Lookups – Chord
• Comparison of DHTs
Peer-to-Peer Networks

• Typically each member stores/provides access to content
• Basically a replication system for files
  • Always a tradeoff between possible location of files and searching difficulty
  • Peer-to-peer allow files to be anywhere → searching is the challenge
  • Dynamic member list makes it more difficult
• What other systems have similar goals?
  • Routing, DNS
The Lookup Problem

Key="title"
Value=MP3 data...
Publisher

Client
Lookup("title")

N1 -> N2 -> N3
N4 -> N5 -> N6
Centralized Lookup (Napster)

Simple, but $O(N)$ state and a single point of failure
Flooded Queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Routed Queries (Chord, etc.)

Publisher
Key="title"
Value=MP3 data...

Client
Lookup("title")
Overview

• P2P Lookup Overview

• Centralized/Flooded Lookups

• Routed Lookups – Chord

• Comparison of DHTs
Centralized: Napster

- Simple centralized scheme $\rightarrow$ motivated by ability to sell/control

- How to find a file:
  - On startup, client contacts central server and reports list of files
  - Query the index system $\rightarrow$ return a machine that stores the required file
    - Ideally this is the closest/least-loaded machine
  - Fetch the file directly from peer
Centralized: Napster

• Advantages:
  • Simple
  • Easy to implement sophisticated search engines on top of the index system

• Disadvantages:
  • Robustness, scalability
  • Easy to sue!
Flooding: Old Gnutella

• On startup, client contacts any servent (server + client) in network
  • Servent interconnection used to forward control (queries, hits, etc)

• Idea: broadcast the request

• How to find a file:
  • Send request to all neighbors
  • Neighbors recursively forward the request
  • Eventually a machine that has the file receives the request, and it sends back the answer
  • Transfers are done with HTTP between peers
Flooding: Old Gnutella

• Advantages:
  • Totally decentralized, highly robust

• Disadvantages:
  • Not scalable; the entire network can be swamped with request (to alleviate this problem, each request has a TTL)
  • Especially hard on slow clients
    • At some point broadcast traffic on Gnutella exceeded 56kbps – what happened?
    • Modem users were effectively cut off!
Flooding: Old Gnutella Details

- Basic message header
  - Unique ID, TTL, Hops
- Message types
  - Ping – probes network for other servents
  - Pong – response to ping, contains IP addr, # of files, # of Kbytes shared
  - Query – search criteria + speed requirement of servent
  - QueryHit – successful response to Query, contains addr + port to transfer from, speed of servent, number of hits, hit results, servent ID
  - Push – request to servent ID to initiate connection, used to traverse firewalls
- Ping, Queries are flooded
- QueryHit, Pong, Push reverse path of previous message
Assume: m1’s neighbors are m2 and m3; m3’s neighbors are m4 and m5;...
Flooding: Gnutella, Kazaa

- Modifies the Gnutella protocol into two-level hierarchy
  - Hybrid of Gnutella and Napster
- Supernodes
  - Nodes that have better connection to Internet
  - Act as temporary indexing servers for other nodes
  - Help improve the stability of the network
- Standard nodes
  - Connect to supernodes and report list of files
  - Allows slower nodes to participate
- Search
  - Broadcast (Gnutella-style) search across supernodes
- Disadvantages
  - Kept a centralized registration → allowed for law suits 😞
Overview

- P2P Lookup Overview
- Centralized/Flooded Lookups
- Routed Lookups – Chord
- Comparison of DHTs
Routing: Structured Approaches

- **Goal:** make sure that an item (file) identified is always found in a reasonable # of steps
- **Abstraction:** a distributed hash-table (DHT) data structure
  - \texttt{insert(id, item)};
  - \texttt{item = query(id)};
  - Note: item can be anything: a data object, document, file, pointer to a file…
- **Proposals**
  - CAN (ICIR/Berkeley)
  - Chord (MIT/Berkeley)
  - Pastry (Rice)
  - Tapestry (Berkeley)
  - …
Routing: Chord

• Associate to each node and item a unique \textit{id} in an \textit{uni}-dimensional space

• Properties
  • Routing table size $O(\log(N))$, where $N$ is the total number of nodes
  • Guarantees that a file is found in $O(\log(N))$ steps
Aside: Hashing

• Advantages
  • Let nodes be numbered 1..m
  • Client uses a *good* hash function to map a URL to 1..m
  • Say hash (url) = x, so, client fetches content from node x
  • No duplication – not being fault tolerant.
• One hop access
• Any problems?
  • What happens if a node goes down?
  • What happens if a node comes back up?
  • What if different nodes have different views?
Robust hashing

- Let 90 documents, node 1..9, node 10 which was dead is alive again
- % of documents in the wrong node?
  - 10, 19-20, 28-30, 37-40, 46-50, 55-60, 64-70, 73-80, 82-90
  - *Disruption coefficient* = $\frac{1}{2}$
  - Unacceptable, use consistent hashing – idea behind Akamai!
Consistent Hash

- “view” = subset of all hash buckets that are visible

- Desired features
  - Balanced – in any one view, load is equal across buckets
  - Smoothness – little impact on hash bucket contents when buckets are added/removed
  - Spread – small set of hash buckets that may hold an object regardless of views
  - Load – across all views # of objects assigned to hash bucket is small
Consistent Hash – Example

• Construction
  • Assign each of C hash buckets to random points on mod $2^n$ circle, where, hash key size = $n$.
  • Map object to random position on circle
  • Hash of object = closest clockwise bucket

• Smoothness → addition of bucket does not cause much movement between existing buckets
• Spread & Load → small set of buckets that lie near object
• Balance → no bucket is responsible for large number of objects
Routing: Chord Basic Lookup

```
N105 --> N120 --> N10
   |       |     |
  N90 ---|     |
     |  \
     |   \
     |    \
     |     \
K80   |     |
     |  \
     |   \
     |    \
     |     \
   N90 --|     |
   |       |     |
   |       |     |
   |       |     |
   |       |     |
   N60
```

"Where is key 80?"

"N90 has K80"
Routing: Finger table - Faster Lookups

N80

1/8
1/16
1/32
1/64
1/128

1/4
1/2
Routing: Chord Summary

- Assume identifier space is $0...2^m$
- Each node maintains
  - Finger table
    - Entry $i$ in the finger table of $n$ is the first node that succeeds or equals $n + 2^i$
  - Predecessor node
- An item identified by $id$ is stored on the successor node of $id$
Routing: Chord Example

- Assume an identifier space 0..8
- Node n1:(1) joins all entries in its finger table are initialized to itself
Routing: Chord Example

- Node n2:(2) joins

### Succ. Table

<table>
<thead>
<tr>
<th>i</th>
<th>id + 2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
Routing: Chord Example

- Nodes n3:(0), n4:(6) join
Routing: Chord Examples

- Nodes: n1:(1), n2(2), n3(0), n4(6)
- Items: f1:(7), f2:(1)
Routing: Query

- Upon receiving a query for item id, a node
- Check whether stores the item locally
- If not, forwards the query to the largest node in its successor table that does not exceed id

```
<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>id+2^i</th>
<th>succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Succ. Table

Items

query(7)
What can DHTs do for us?

- Distributed object lookup
  - Based on object ID
- De-centralized file systems
  - CFS, PAST, Ivy
- Application Layer Multicast
  - Scribe, Bayeux, Splitstream
- Databases
  - PIER
Overview

• P2P Lookup Overview

• Centralized/Flooded Lookups

• Routed Lookups – Chord

• Comparison of DHTs
Comparison

- Many proposals for DHTs
  - Tapestry (UCB) -- Symphony (Stanford) -- 1hop (MIT)
  - Pastry (MSR, Rice) -- Tangle (UCB) -- conChord (MIT)
  - Chord (MIT, UCB) -- SkipNet (MSR, UW) -- Apocrypha (Stanford)
  - CAN (UCB, ICSI) -- Bamboo (UCB) -- LAND (Hebrew Univ.)
  - Viceroy (Technion) -- Hieras (U.Cinn) -- ODRI (Texas A&M)
  - Kademlia (NYU) -- Sprout (Stanford)
  - Kelips (Cornell) -- Calot (Rochester)
  - Koorde (MIT) -- JXTA’s (Sun)

- What are the right design choices? Effect on performance?
Deconstructing DHTs

Two observations:

1. Common approach
   - N nodes; each labeled with a virtual identifier (128 bits)
   - define “distance” function on the identifiers
   - routing works to reduce the distance to the destination

2. DHTs differ primarily in their definition of “distance”
   - typically derived from (loose) notion of a routing geometry
DHT Routing Geometries

- **Geometries:**
  - Tree (Plaxton, Tapestry)
  - Ring (Chord)
  - Hypercube (CAN)
  - XOR (Kademlia)
  - Hybrid (Pastry)

- **What is the impact of geometry on routing?**
Tree (Plaxton, Tapestry)

Geometry

- nodes are leaves in a binary tree
- distance = height of the smallest common subtree
- logN neighbors in subtrees at distance 1, 2, ..., logN
Hypercube (CAN)

Geometry

• nodes are the corners of a hypercube
• distance = #matching bits in the IDs of two nodes
• $\log N$ neighbors per node; each at distance=1 away
Ring (Chord)

Geometry

- nodes are points on a ring
- distance = numeric distance between two node IDs
- logN neighbors exponentially spaced over 0…N
Hybrid (Pastry)

Geometry:
- combination of a tree and ring
- two distance metrics
- default routing uses tree; fallback to ring under failures
  - neighbors picked as on the tree
XOR (Kademlia)

Geometry:

- \text{distance}(A,B) = A \text{ XOR } B
- \log N \text{ neighbors per node spaced exponentially}
- \textbf{not} a ring because there is no single consistent ordering of all the nodes
Geometry’s Impact on Routing

• Routing
  • Neighbor selection: how a node picks its routing entries
  • Route selection: how a node picks the next hop

• Proposed metric: **flexibility**
  • amount of freedom to choose neighbors and next-hop paths
    • **FNS**: flexibility in neighbor selection
    • **FRS**: flexibility in route selection
  • intuition: captures ability to “tune” DHT performance
  • single predictor metric dependent only on routing issues
• Chord algorithm picks $i^{th}$ neighbor at $2^i$ distance
• A different algorithm picks $i^{th}$ neighbor from $[2^i, 2^{i+1})$
FRS for Ring Geometry

- Chord algorithm picks neighbor closest to destination
- A different algorithm picks the best of alternate paths
## Flexibility: at a Glance

<table>
<thead>
<tr>
<th>Flexibility</th>
<th>Ordering of Geometries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbors (FNS)</td>
<td>Hypercube &lt;&lt; Tree, XOR, Ring, Hybrid</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>Routes (FRS)</td>
<td>Tree &lt;&lt; XOR, Hybrid &lt; Hypercube &lt; Ring</td>
</tr>
<tr>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td></td>
<td>(logN/2)</td>
</tr>
<tr>
<td></td>
<td>(logN/2)</td>
</tr>
<tr>
<td></td>
<td>(logN)</td>
</tr>
</tbody>
</table>
Validate over three performance metrics:
1. resilience
2. path latency
3. path convergence

Metrics address two typical concerns:
• ability to handle node failure
• ability to incorporate proximity into overlay routing
Does flexibility affect static resilience?

Tree \ll XOR \approx Hybrid < Hypercube < Ring

*Flexibility in Route Selection matters for Static Resilience*
Which is more effective, FNS or FRS?

![Graph showing CDF of Latency (msec) for different selection methods: Plain Ring, PNS Ring, PRS Ring, and PNS+PRS Ring.]

Plain $\ll$ FRS $\ll$ FNS $\approx$ FNS+FRS

Neighbor Selection is much better than Route Selection
Understanding DHT Routing: Conclusion

• What makes for a “good” DHT?
  • one answer: a flexible routing geometry

• Result: Ring is most flexible
Next Lecture

• DNS, Web and P2P
• Required readings
  • Peer-to-Peer Systems
  • Do incentives build robustness in BitTorrent?