Internet Technology
05r. Distributed Hash Tables

Paul Krzyzanowski
Rutgers University
Spring 2013
Locating content

• Our discussion on peer-to-peer applications focused on content distribution
  – Content was fully distributed

• How do we find the content?

<table>
<thead>
<tr>
<th></th>
<th>Central server (hybrid architecture)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>Network flooding</td>
</tr>
<tr>
<td></td>
<td>Optimized to flood supernodes … but it’s still flooding</td>
</tr>
<tr>
<td>Gnutella &amp; Kazaa</td>
<td>Nothing!</td>
</tr>
<tr>
<td></td>
<td>It’s somebody else’s problem</td>
</tr>
</tbody>
</table>

• Can we do better?
What’s wrong with flooding?

• Some nodes are not always up and some are slower than others
  – Gnutella & Kazaa dealt with this by classifying some nodes as “supernodes” (called “ultrapeers” in Gnutella)

• Poor use of network resources

• Potentially high latency
  – Requests get forwarded from one machine to another
  – Back propagation (e.g., Gnutella design), where the replies go through the same chain of machines used in the query, increases latency even more
Hash tables

• Remember hash functions & hash tables?
  – Linear search: $O(N)$
  – Tree: $O(\log N)$
  – Hash table: $O(1)$
What’s a hash function? (refresher)

• Hash function
  – A function that takes a variable length input (e.g., a string) and generates a (usually smaller) fixed length result (e.g., an integer)
  – Example: hash strings to a range 0-6:
    • \( \text{hash(“Newark’’)} \rightarrow 1 \)
    • \( \text{hash(“Jersey City’’)} \rightarrow 6 \)
    • \( \text{hash(“Paterson’’)} \rightarrow 2 \)

• Hash table
  – Table of \((\text{key, value})\) tuples
  – Look up a key:
    • Hash function maps \text{keys} to a range \(0 \ldots N-1\)
      table of \(N\) elements
      \(i = \text{hash(key)}\)
      \(\text{table}[i]\) contains the item
  – No need to search through the table!
Considerations with hash tables (refresher)

- Picking a good hash function
  - We want uniform distribution of all values of key over the space $0 \ldots N-1$

- Collisions
  - Multiple keys may hash to the same value
    - hash(“Paterson”) $\rightarrow 2$
    - hash(“Edison”) $\rightarrow 2$
  - $\text{table}[i]$ is a bucket (slot) for all such (key, value) sets
  - Within $\text{table}[i]$, use a linked list or another layer of hashing

- Think about a hash table that grows or shrinks
  - If we add or remove buckets $\rightarrow$ need to rehash keys and move items
Distributed Hash Tables (DHT)

- Create a peer-to-peer version of a \((key, value)\) database

- How we want it to work
  1. A peer queries the database with a key
  2. The database finds the peer that has the value
  3. That peer returns the \((key, value)\) pair to the querying peer

- Make it efficient!
  - A query should not generate a flood!

- We’ll look at one DHT implementation called **Chord**
The basic idea

• Each node (peer) is identified by an integer in the range \([0, 2^{n}-1]\)
• Each key is hashed into the range \([0, 2^{n}-1]\)
• Each peer will be responsible for specific keys
  – A key is stored at the closest successor node
  – This is the first node whose ID \(\geq\) hash(key)

• If we arrange the peers in a logical ring (incrementing IDs) then a peer needs to know only of its successor and predecessor
  – This limited knowledge of peers makes it an overlay network
Key assignment

• Example: $n=16$; system with 4 nodes (so far)

• A key is stored at a successor
  – a node whose value is $\geq$ hash(key)

Node 14 is responsible for keys 11, 12, 13, 14

Node 3 is responsible for keys 15, 0, 1, 2, 3

Node 8 is responsible for keys 4, 5, 6, 7, 8

Node 10 is responsible for keys 9, 10

No nodes at these empty positions
Handling requests

• Any peer can get a request (insert or query). If the hash(key) is not for its ranges of keys, it forwards the request to a successor.

• The process continues until the responsible node is found
  – Worst case: with $p$ nodes, traverse $p-1$ nodes; that’s $O(N)$ (yuck!)
  – Average case: traverse $p/2$ nodes (still yuck!)

Node 14 is responsible for keys 11, 12, 13, 14
Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 10 is responsible for keys 9, 10
Node 8 is responsible for keys 4, 5, 6, 7, 8

Node #10 can process the request

Query( hash(key)=9 )
Let’s figure out three more things

1. Adding/removing nodes
2. Improving lookup time
3. Fault tolerance
Adding a node

- Some keys that were assigned to a node’s successor now get assigned to the new node
- Data for those \((key, value)\) pairs must be moved to the new node

Node 14 is responsible for keys 11, 12, 13, 14

Node 10 is responsible for keys 9, 10

New node added: ID = 6

Node 8 was responsible for keys 4, 5, 6, 7, 8
Now it’s responsible for keys 7, 8

Node 6 is responsible for keys 4, 5, 6

Node 3 is responsible for keys 15, 0, 1, 2, 3
Removing a node

- Keys are reassigned to the node’s successor
- Data for those \((key, value)\) pairs must be moved to the successor

Node 14 was responsible for keys 11, 12, 13, 14
Node 14 is now responsible for keys 9, 10, 11, 12, 13, 14
Node 10 was responsible for keys 9, 10
Node 10 removed

Node 3 is responsible for keys 15, 0, 1, 2, 3
Node 6 is responsible for keys 4, 5, 6
Node 8 is responsible for keys 7, 8

Node 10 removed
Performance

• We’re not thrilled about $O(N)$ lookup

• Simple approach for great performance
  – Have all nodes know about each other
  – When a peer gets a node, it searches its table of nodes for the node that owns those values
  – Gives us $O(1)$ performance
  – Add/remove node operations must inform everyone
  – Not a good solution if we have millions of peers (huge tables)
Finger tables

• Compromise to avoid huge per-node tables
  – Use finger tables to place an upper bound on the table size
• Finger table = partial list of nodes
• At each node, $i^{\text{th}}$ entry in finger table identifies node that succeeds it by at least $2^{i-1}$ in the circle
  – finger_table[0]: immediate (1$^{\text{st}}$) successor
  – finger_table[1]: successor after that (2$^{\text{nd}}$)
  – finger_table[2]: 4$^{\text{th}}$ successor
  – finger_table[3]: 8$^{\text{th}}$ successor
  – …
• $O(\log N)$ nodes need to be contacted to find the node that owns a key
  … not as cool as $O(1)$ but way better than $O(N)$
Fault tolerance

• Nodes might die
  – (key, value) data would need to be replicated
  – Create R replicas, storing each one at $R-1$ successor nodes in the ring

• It gets a bit complex
  – A node needs to know how to find its successor’s successor (or more)
    • Easy if it knows all nodes!
  – When a node is back up, it needs to check with successors for updates
  – Any changes need to be propagated to all replicas
The end