Overview

• What problems do DHTs solve?
• How are DHTs implemented?
A hash table is a data structure that stores (key, object) pairs. Key is mapped to a table index via a hash function for fast lookup.

Content distribution networks
- Given an URL, returns the object
Example of a Hash table: a web cache

<table>
<thead>
<tr>
<th><a href="http://www.cnn.com">http://www.cnn.com</a></th>
<th>Page content</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.nytimes.com">http://www.nytimes.com</a></td>
<td>....</td>
</tr>
<tr>
<td><a href="http://www.slashdot.org">http://www.slashdot.org</a></td>
<td>...</td>
</tr>
<tr>
<td>... 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Client requests [http://www.cnn.com](http://www.cnn.com)
- Web cache returns the page content located at the 1\textsuperscript{st} entry of the table.
DHT: why?

• If the number of objects is large, it is impossible for any single node to store it.

• Solution: distributed hash tables.
  – Split one large hash table into smaller tables and distribute them to multiple nodes
DHT
A content distribution network

- A single provider that manages multiple replicas.
- A client obtains content from a close replica.
Basic function of DHT

• DHT is a “virtual” hash table
  – Input: a key
  – Output: a data item
• Data Items are stored by a network of nodes.
• DHT abstraction
  – Input: a key
  – Output: the node that stores the key
• Applications handle key and data item association.
DHT: a visual example

Insert \((K_1, V_1)\)
DHT: a visual example

Retrieve \( K_1 \)
Desired properties of DHT

• Scalability: each node does not keep much state

• Performance: look up latency is small

• Load balancing: no node is overloaded with a large amount of state

• Dynamic reconfiguration: when nodes join and leave, the amount of state moved from nodes to nodes is small.

• Distributed: no node is more important than others.
A straw man design

- Suppose all keys are integers.
- The number of nodes in the network is $n$.
- $id = \text{key} \mod n$
When node 2 dies

- A large number of data items need to be rehashed.
Fix: consistent hashing

• When a node joins or leaves, the expected fraction of objects that must be moved is the minimum needed to maintain a balanced load.

• A node is responsible for a range of keys

• All DHTs implement consistent hashing
Chord: basic idea

• Hash both node id and key into a m-bit one-dimension circular identifier space

• Consistent hashing: a key is stored at a node whose identifier is closest to the key in the identifier space
  – Key refers to both the key and its hash value.
Basic components of DHTs

• Overlapping key and node identifier space
  – Hash(www.cnn.com/image.jpg) → a n-bit binary string
  – Nodes that store the objects also have n-bit string as their identifiers

• Building routing tables
  – Next hops
  – Distance functions
  – These two determine the geometry of DHTs
    • Ring, Tree, Hybercubes, hybrid (tree + ring) etc.
  – Handle node join and leave

• Lookup and store interface
A key is stored at its successor: node with next higher ID
Chord: how to find a node that stores a key?

- Solution 1: every node keeps a routing table to all other nodes
  - Given a key, a node knows which node id is successor of the key
  - The node sends the query to the successor
  - What are the advantages and disadvantages of this solution?
Solution 2: every node keeps a routing entry to the node’s successor (a linked list)
Simple lookup algorithm

Lookup(my-id, key-id)

1. n = my successor
2. if my-id < n < key-id
   - call Lookup(key-id) on node n    // next hop
3. else
   - return my successor          // done

- Correctness depends only on successors
- Q1: will this algorithm miss the real successor?
- Q2: what’s the average # of lookup hops?
Solution 3: “Finger table” allows log(N)-time lookups

- Analogy: binary search

![Diagram with arrows pointing to fractions 1/8, 1/16, 1/32, 1/64, 1/128 towards N80]
Finger $i$ points to successor of $n + 2^{i-1}$

- A finger table entry includes Chord Id and IP address
- Each node stores a small table $\log(N)$
Chord finger table example

Keys:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
<td>0</td>
</tr>
</tbody>
</table>

Keys: 5,6

Keys:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>[5,1)</td>
<td>0</td>
</tr>
</tbody>
</table>

Keys: 1

Keys:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>[4,5)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
<td>0</td>
</tr>
</tbody>
</table>

Keys: 2

Diagram:

8 notes labeled 0 through 7 with finger assignments and chord overlap intervals.
Lookup with fingers

Lookup(my-id, key-id)
   look in local finger table for
      highest node n s.t. my-id < n < key-id
   if n exists
      call Lookup(key-id) on node n  // next hop
   else
      return my successor   // done
// ask node n to find the successor of id
n.find_successor(id)
   if (id ∈ (n, successor])
      return successor;
   else
      n' = closest_preceding_node(id);
      return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
   for i = m downto 1
      if (finger[i] ∈ (n, id))
         return finger[i];
   return n;
Chord lookup example

- Lookup(1,6)
- Lookup(1,2)
Node join

• Maintain the invariant
  1. Each node’s successor is correctly maintained
  2. For every node k, node successor(k) answers for k. It’s desirable that finger table entries are correct

• Each nodes maintains a predecessor pointer

• Tasks:
  – Initialize predecessor and fingers of new node
  – Update existing nodes’ state
  – Notify apps to transfer state to new node
Chord Joining: linked list insert

- Node n queries a known node n’ to initialize its state
- for its successor: lookup (n)
2. N36 sets its own successor pointer
Join (3)

3. Copy keys 26..36 from N40 to N36

• Note that join does not make the network aware of n
Join (4): stabilize

- Stabilize 1) obtains a node n’s successor’s predecessor x, and determines whether x should be n’s successor 2) notifies n’s successor n’s existence
  - N25 calls its successor N40 to return its predecessor
  - Set its successor to N36
  - Notifies N36 it is predecessor

- Update finger pointers in the background periodically
  - Find the successor of each entry i
- Correct successors produce correct lookups
Failures might cause incorrect lookup

N80 doesn’t know correct successor, so incorrect lookup
Solution: successor lists

• Each node knows $r$ immediate successors
• After failure, will know first live successor
• Correct successors guarantee correct lookups

• Guarantee is with some probability

• Higher layer software can be notified to duplicate keys at failed nodes to live successors
Choosing the successor list length

• Assume 1/2 of nodes fail

• $P(\text{successor list all dead}) = (1/2)^r$
  – I.e. $P(\text{this node breaks the Chord ring})$
  – Depends on independent failure

• $P(\text{no broken nodes}) = (1 - (1/2)^r)^N$
  – $r = 2\log(N)$ makes prob. = $1 - 1/N$
Lookup with fault tolerance

Lookup(my-id, key-id)
  look in local finger table and successor-list
  for highest node n s.t. my-id < n < key-id
  if n exists
    call Lookup(key-id) on node n // next hop
  if call failed,
    remove n from finger table
  return Lookup(my-id, key-id)
else return my successor // done
Chord performance

• Per node storage
  – Ideally: K/N
  – Implementation: large variance due to unevenly node id distribution

• Lookup latency
  – $O(\log N)$
Comments on Chord

• DHTs are used for p2p file lookup in the real world

• ID distance ≠ Network distance
  – Reducing lookup latency and locality are research challenges

• Strict successor selection
  – Can’t overshoot

• Asymmetry
  – A node does not learn its routing table entries from queries it receives
Conclusion

• Consistent Hashing
  – What problem does it solve

• Design of DHTs
  – Chord: ring

• Kademlia: tree
  – Used in practice, emule, Bittorrent
  – CAN: hybercube
  – Much more others: Pastry, Tapestry, Viceroy....
Discussion

• What tradeoff does chord make?
• How can we improve chord’s lookup latency?
• What are the possible applications of DHT?
• Recursive lookup or iterative lookup?