Distributed Systems

17. Distributed Lookup

Paul Krzyzanowski
Rutgers University
Fall 2016

November 5, 2016
© 2014-2016 Paul Krzyzanowski

Distributed Lookup

- Look up (key, value)
- Cooperating set of nodes

Ideally:
- No central coordinator
- Some nodes can be down

Approaches

1. Central coordinator
   - Napster

2. Flooding
   - Gnutella

3. Distributed hash tables
   - CAN, Chord, Amazon Dynamo, Tapestry …

1. Central Coordinator

- Example: Napster
- Central directory
  - Identifies content (names) and the servers that host it
  - `lookup(name) → (list of servers)`
  - Download from any of available servers
  - Pick the best one by pinging and comparing response times

1. Central Coordinator - Napster

- Pros
  - Super simple
  - Search is handled by a single server (master)
  - The directory server is a single point of control
    - Provides definitive answers to a query

- Cons
  - Master has to maintain state of all peers
  - Server gets all the queries
  - The directory server is a single point of control
    - No directory, no service!

1. Central Coordinator

- Another example: GFS
  - Controlled environment compared to Napster
  - Content for a given key is broken into chunks
  - Master handles all queries … but not the data
2. Query Flooding

- Example: Gnutella distributed file sharing

- Well-known nodes act as anchors
  - Nodes with files inform an anchor about their existence
  - Nodes select other nodes as peers

- Send a query to peers if a file is not present locally
  - Each request contains:
    - Query key
    - Unique request ID
    - Time to Live (TTL, maximum hop count)

- Peer either responds or routes the query to its neighbors
  - Repeat until TTL = 0 or if the request ID has been processed
  - If found, send response (node address) to the requestor
  - Back propagation: series of responses reaches originator

Overlay network

An overlay network is a virtual network formed by peer connections
  - Any node might know about a small set of machines
  - “Neighbors” may not be physically close to you

Flooding Example: Overlay Network

Flooding Example: Query Flood
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., in Gnutella's design), where the replies go through the same chain of machines used in the query, increases latency even more

3. Distributed Hash Tables

Locating content

- How do we locate distributed content?
  - A central server is the easiest

<table>
<thead>
<tr>
<th>System</th>
<th>Location Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>Central server</td>
</tr>
<tr>
<td>Gnutella &amp; Kazaa</td>
<td>Network flooding</td>
</tr>
<tr>
<td>BitTorrent</td>
<td>Nothing!</td>
</tr>
</tbody>
</table>

- Can we do better?

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 0-7:
    • hash("Newark") → 1
    • hash("Jersey City") → 6
    • hash("Paterson") → 2

• Table
  – Table of (key, value) tuples
  – Look up a key:
    • Hash function maps keys to a range 0 … N-1
    • 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 … N-1

• Collisions
  – Multiple keys may hash to the same value
    • hash("Paterson") → 2
    • hash("Edison") → 2
  – Table[i] is a bucket (slot) for all such (key, value) sets
  – Within 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 → need to rehash keys and move items

Distributed Hash Tables (DHT)

• Create a peer-to-peer version of a (key, value) data store

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

• Make it efficient!
  – A query should not generate a flood!

Distributed Hashing Case Study

CAN: Content Addressable Network

3. Distributed hashing

• Spread the hash table across multiple nodes
  • Each node stores a portion of the key space
  • lookup(key) → node ID that holds (key, value)

• Questions
  How do we partition the data & do the lookup?
  & keep the system decentralized?
  & make the system scalable (lots of nodes)?
  & fault tolerant (replicated data)?
**CAN design**

- Create a logical grid
  - \( x \cdot y \) in 2-D but not limited to 2-D

- Separate hash function per dimension
  - \( h_x(key), h_y(key) \)

- A node;
  - Is responsible for a range of values in each dimension
  - Knows its neighboring nodes

**CAN partitioning**

Any node can be split in two – either horizontally or vertically

Associated data has to be moved to the new node based on \( \text{hash}(key) \)

Neighbors need to be made aware of the new node

A node knows only of its neighbors

**CAN key→node mapping: 2 nodes**

\[
\begin{align*}
    x &= \text{hash}_x(key) \\
    y &= \text{hash}_y(key) \\
    \text{if } x < \left( \frac{x_{\text{max}}}{2} \right) \\
        &\quad n_1 \text{ has (key, value)} \\
    \text{if } x \geq \left( \frac{x_{\text{max}}}{2} \right) \\
        &\quad n_2 \text{ has (key, value)} \\
\end{align*}
\]

\( n_2 \) is responsible for a zone \( x=\left( \frac{x_{\text{max}}}{2} \right) \ldots x_{\text{max}} \), \( y=0 \ldots y_{\text{max}} \)

**CAN key→node mapping: 2 nodes**

\[
\begin{align*}
    x &= \text{hash}_x(key) \\
    y &= \text{hash}_y(key) \\
    \text{if } x < \left( \frac{x_{\text{max}}}{2} \right) \\
        &\quad n_0 \text{ has (key, value)} \\
    \text{if } x \geq \left( \frac{x_{\text{max}}}{2} \right) \\
        &\quad n_2 \text{ has (key, value)} \\
\end{align*}
\]

**CAN neighbors**

Neighbors refer to nodes that share adjacent zones in the overlay network

\( n_4 \) only needs to keep track of \( n_5, n_6, n_8 \) as its right neighbor.
Distributed Systems

CAN routing

- **lookup(key)** on a node that does not own the value
- Compute \( \text{hash}(\text{key}) \) and route request to a neighboring node
- Ideally: route to minimize distance to destination

**Performance**
- For \( n \) nodes in \( d \) dimensions
- \( 2^d \) neighbors
- Average route for 2 dimensions = \( O(\sqrt{n}) \) hops

**To handle failures**
- Share knowledge of neighbor’s neighbors
- One of the node’s neighbors takes over the failed zone

Chord & consistent hashing

- A key is hashed to an \( m \) bit value: \( 0 \ldots (2^m-1) \)
- A logical ring is constructed for the values \( 0 \ldots (2^m-1) \)
- Nodes are placed on the ring at \( \text{hash}(\text{IP address}) \)

**Key assignment**

- Example: \( n=16 \); system with 4 nodes (so far)
- Key value data is stored at a **successor**
  - a node whose value is \( \geq \text{hash(key)} \)

**Handling query 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: traverse \( p \) nodes, traverse \( p/2 \) nodes
  - Average case: traverse \( p/2 \) nodes

Distributed Hashing Case Study

**Chord**

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

- No nodes at these empty positions

Node hash(IP address) = 3

Query (hash(key) = 9)

Node #10 can process the request

Node #14 can process the request
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

Removing a node

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

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
• Need to know successors
  – A node needs to know how to find its successor’s successor (or more)
  – When a node is back up, it needs to check with successors for updates
  – Any changes need to be propagated to all replicas

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
      – Maybe not a good solution if we have millions of peers (huge tables)

Finger tables

• Compromise to avoid large tables at each node
  – Use finger tables to place an upper bound on the table size
  – Finger table = partial list of nodes
  – At each node, $i^{th}$ entry in finger table identifies node that succeeds it by at least $2^{i-1}$ in the circle
    – finger_table[0]: immediate (1st) successor
    – finger_table[1]: successor after that (2nd)
    – finger_table[2]: 4th successor
    – finger_table[3]: 8th 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 $O(N)$
Improving performance even more

• Let’s revisit O(1) lookup
• Each node keeps track of all current nodes in the group
  – Is that really so bad?
  – We might have thousands of nodes … so what?
• Any node will now know which node holds a \((key, value)\)
• Add or remove a node: send updates to all other nodes

The end