CSE 486/586 Distributed Systems
Distributed Hash Tables

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last Time
• Evolution of peer-to-peer
  – Central directory (Napster)
  – Query flooding (Gnutella)
  – Hierarchical overlay (Kazaa, modern Gnutella)
• BitTorrent
  – Focuses on parallel download
  – Prevents free-riding

Today's Question
• How do we organize the nodes in a distributed system?
  • Up to the 90’s
    – Prevalent architecture: client-server (or master-slave)
      – Unequal responsibilities
  • Now
    – Emerged architecture: peer-to-peer
      – Equal responsibilities
  • Today: studying peer-to-peer as a paradigm

What We Want
• Functionality: lookup-response

What We Don't Want
• Cost (scalability) & no guarantee for lookup

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Lookup Latency</th>
<th>#Messages for a lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>(O(N)@server)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gnutella</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>(worst case)</td>
<td></td>
<td>(worst case)</td>
<td>(worst case)</td>
</tr>
</tbody>
</table>

• Napster: cost not balanced, too much for the server-side
• Gnutella: cost still not balanced, just too much, no guarantee for lookup

What We Want
• What data structure provides lookup-response?
• Hash table: data structure that associates keys with values

- Name-value pairs (or key-value pairs)
  – E.g., "BritneyHitMe.mp3" and "12.78.183.2"
Hashing Basics

- **Hash function**
  - Function that maps a large, possibly variable-sized datum into a small datum, often a single integer that serves to index an associative array
  - In short: maps n-bit datum into k buckets \( (k \ll 2^n) \)
  - Provides time- & space-saving data structure for lookup
- **Main goals:**
  - Low cost
  - Deterministic
  - Uniformity (load balanced)
- **E.g., mod**
  - \( k \) buckets \( (k \ll 2^n) \), data \( d \) (n-bit)
  - \( b = d \mod k \)
  - Distributes load uniformly only when data is distributed uniformly

DHT: Goal

- Let’s build a distributed system with a hash table abstraction!

Where to Keep the Hash Table

- **Server-side** → Napster
- **Client-local** → Gnutella
- What are the requirements (think Napster and Gnutella)?
  - Deterministic lookup
  - Low lookup time (shouldn’t grow linearly with the system size)
  - Should balance load even with node join/leave
- What we’ll do: partition the hash table and distribute them among the nodes in the system
- We need to choose the right hash function
- We also need to somehow partition the table and distribute the partitions with minimal relocation of partitions in the presence of join/leave

Using Basic Hashing and Bucket Partitioning?

- Hashing: Suppose we use modulo hashing
  - Number servers 1..k
- Partitioning: Place \( X \) on server \( i = (X \mod k) \)
  - Problem? Data may not be uniformly distributed

Where to Keep the Hash Table

- Consider problem of data partition:
  - Given document \( X \), choose one of \( k \) servers to use
  - Key can be the filename and value can be the document itself.
- **Two-level mapping**
  - Hashing: Map one (or more) key(s) to a hash value (the distribution should be balanced)
  - Partitioning: Map a hash value to a server (each server load should be balanced even with node join/leave)
- Let’s look at a simple approach and think about pros and cons.
  - Hashing with mod, and partitioning with buckets
**CSE 486/586 Administrivia**

- PA2-B due on Friday next week, 3/13
- (In class) Midterm on Wednesday (3/11)
  - 1-page cheat sheet (front and back)
- Mid-semester course evaluation is up. Please participate.
- No office hours with Steve today.
- PA2-A grades are posted. Re-grading this week.

**Chord DHT**

- A distributed hash table system using consistent hashing
- Organizes nodes in a ring
- Maintains neighbors for correctness and shortcuts for performance
- DHT in general
  - DHT systems are "structured" peer-to-peer as opposed to "unstructured" peer-to-peer such as Napster, Gnutella, etc.
  - Used as a base system for other systems, e.g., many "trackerless" BitTorrent clients, Amazon Dynamo, distributed repositories, distributed file systems, etc.
- It shows an example of principled design.

**Chord Ring: Global Hash Table**

- Represent the hash key space as a virtual ring
  - A ring representation instead of a table representation.
- Use a hash function that evenly distributes items over the hash space, e.g., SHA-1
- Map nodes (buckets) in the same ring
- Used in DHTs, memcached, etc.

**Chord: Consistent Hashing**

- Partitioning: Maps data items to its “successor” node
- Advantages
  - Even distribution
  - Few changes as nodes come and go...

**Chord: Node Organization**

- Maintain a circularly linked list around the ring
  - Every node has a predecessor and successor
- Separate join and leave protocols
### Chord: Basic Lookup

```c
lookup (id):
  if ( id > pred.id && id <= my.id )
    return my.id;
  else
    return succ.lookup(id);
```

- Route hop by hop via successors
  - $O(n)$ hops to find destination id

---

### Chord: Efficient Lookup

- $i$th entry at peer with id $n$ is first peer with:
  - $id \geq n + 2^i(m + 1)$

**Finger Table at N80**

```
<table>
<thead>
<tr>
<th>i</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>114</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>
```

- Route greedily via distant "finger" nodes
  - $O(\log n)$ hops to find destination id

---

### Chord: Node Joins and Leaves

- When a node joins
  - Node does a lookup on its own id
  - And learns the node responsible for that id
  - This node becomes the new node’s successor
  - And the node can learn that node’s predecessor (which will become the new node’s predecessor)

- Monitor
  - If doesn’t respond for some time, find new

- Leave
  - Clean (planned) leave: notify the neighbors
  - Unclean leave (failure): need an extra mechanism to handle lost (key, value) pairs, e.g., as Dynamo does.

---

### Summary

- **DHT**
  - Gives a hash table as an abstraction
  - Partitions the hash table and distributes them over the nodes
  - "Structured" peer-to-peer

- **Chord DHT**
  - Based on consistent hashing
  - Balances hash table partitions over the nodes
  - Basic lookup based on successors
  - Efficient lookup through fingers

---
Acknowledgements

• These slides contain material developed and copyrighted by Indranil Gupta (UIUC), Michael Freedman (Princeton), and Jennifer Rexford (Princeton).