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

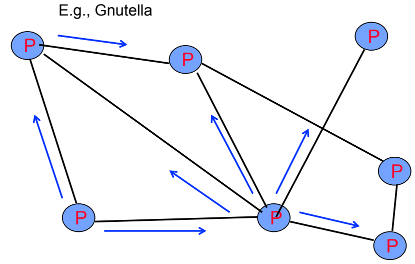
Today's Question

- 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
- Studying an example of client-server: DNS
- Today: studying peer-to-peer as a paradigm

What We Want

- Functionality: lookup-response

E.g., Gnutella



What We Don't Want

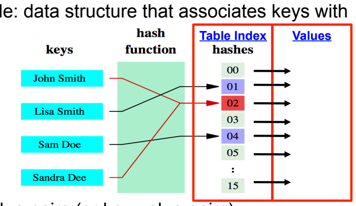
- Cost (scalability) & no guarantee for lookup

	Memory	Lookup Latency	#Messages for a lookup
Napster	*O(1)* (*O(N)*@server)	*O(1)*	*O(1)*
Gnutella	*O(N)*	*O(N)*	*O(N)*

- 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



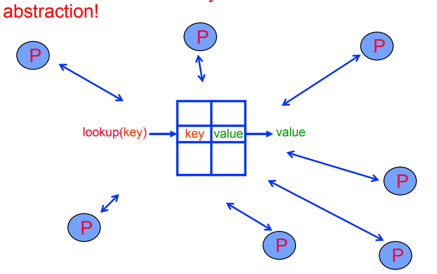
- Name-value pairs (or key-value pairs)
  - E.g., "http://www.cnn.com/foo.html" and the Web page
  - E.g., "BritneyHitMe.mp3" and "12.78.183.2"

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 ($k << 2^n$)
  - Provides time- & space-saving data structure for lookup
- Main goals:
  - Low cost
  - Deterministic
  - Uniformity (load balanced)
- E.g., mod
  - k buckets ($k << 2^n$), data d (*n-bit*)
  - $b = d \bmod k$
  - 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

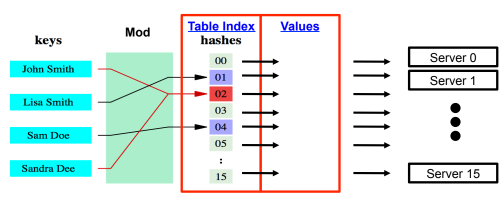
- Server-side → Napster
- Client-local → Gnutella
- What are the requirements?
  - Deterministic lookup
  - Low lookup time (shouldn't grow linearly with the system size)
  - Should balance load even with node join/leave
- What we'll do: partition the hash table and distribute them among the nodes in 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

Where to Keep the Hash Table

- Consider problem of data partition:
  - Given document X, choose one of k servers to use
- Two-level mapping
  - Map one (or more) data item(s) to a hash value (the distribution should be balanced)
  - Map a hash value to a server (each server load should be balanced even with node join/leave)

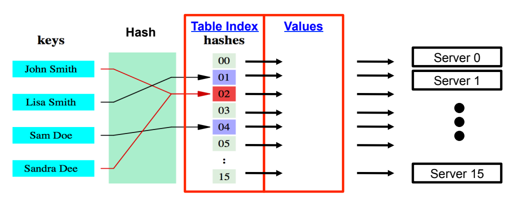
Using Basic Hashing?

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



Using Basic Hashing?

- Place X on server $i = hash(X) \bmod k$
- Problem?
  - What happens if a server fails or joins ($k \rightarrow k\pm1$)?
  - Answer: (Almost) all entries get remapped to new nodes!



CSE 486/586 Administrivia

- PA2-B due in 2 weeks.
- (In class) Midterm on Wednesday (3/11)

Chord DHT

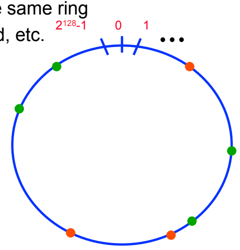
- A distributed hash table system using consistent hashing
- Organizes nodes in a ring
- Maintains neighbors for correctness and shortcuts for performance
- DHT in general
  – DHT systems are "structured" peer-to-peer as opposed to "unstructured" peer-to-peer such as Napster, Gnutella, etc.
  – Used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

Chord: Consistent Hashing

- Represent the hash key space as a ring
- 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.

Id space represented as a ring.

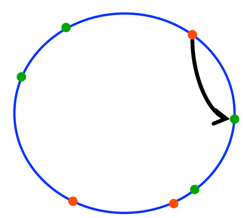
Hash(name) → object_id
Hash(IP_address) → node_id



Chord: Consistent Hashing

- Maps data items to its "successor" node
- Advantages
  – Even distribution
  – Few changes as nodes come and go…

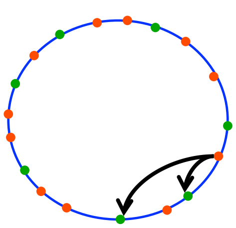
Hash(name) → object_id
Hash(IP_address) → node_id



Chord: When nodes come and go…

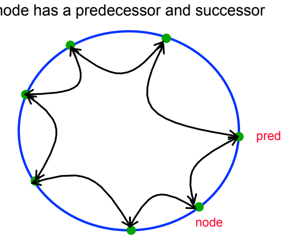
- Small changes when nodes come and go
  – Only affects mapping of keys mapped to the node that comes or goes

Hash(name) → object_id
Hash(IP_address) → node_id



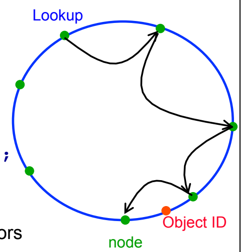
Chord: Node Organization

- Maintain a circularly linked list around the ring
  – Every node has a predecessor and successor



Chord: Basic Lookup

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



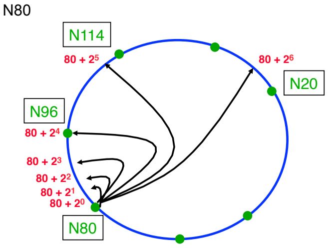
- Route hop by hop via successors
  - O(n) hops to find destination id

Chord: Efficient Lookup --- Fingers

- *i*th entry at peer with id *n* is first peer with:
  - id >= $n + 2^i (\bmod 2^m)$

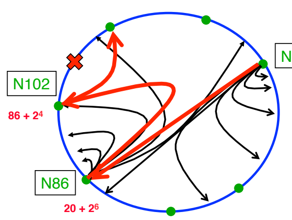
Finger Table at N80

i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	114
6	20



Finger Table

- Finding a <key, value> using fingers



Chord: Efficient Lookup --- Fingers

```
lookup (id):
  if ( id > pred.id &&
       id <= my.id )
    return my.id;
  else
    // fingers() by decreasing distance
    for finger in fingers():
      if id >= finger.id
        return finger.lookup(id);
    return succ.lookup(id);
```

- 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

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).