CSE 486/586 Distributed Systems

Distributed Hash Tables

Slides by 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

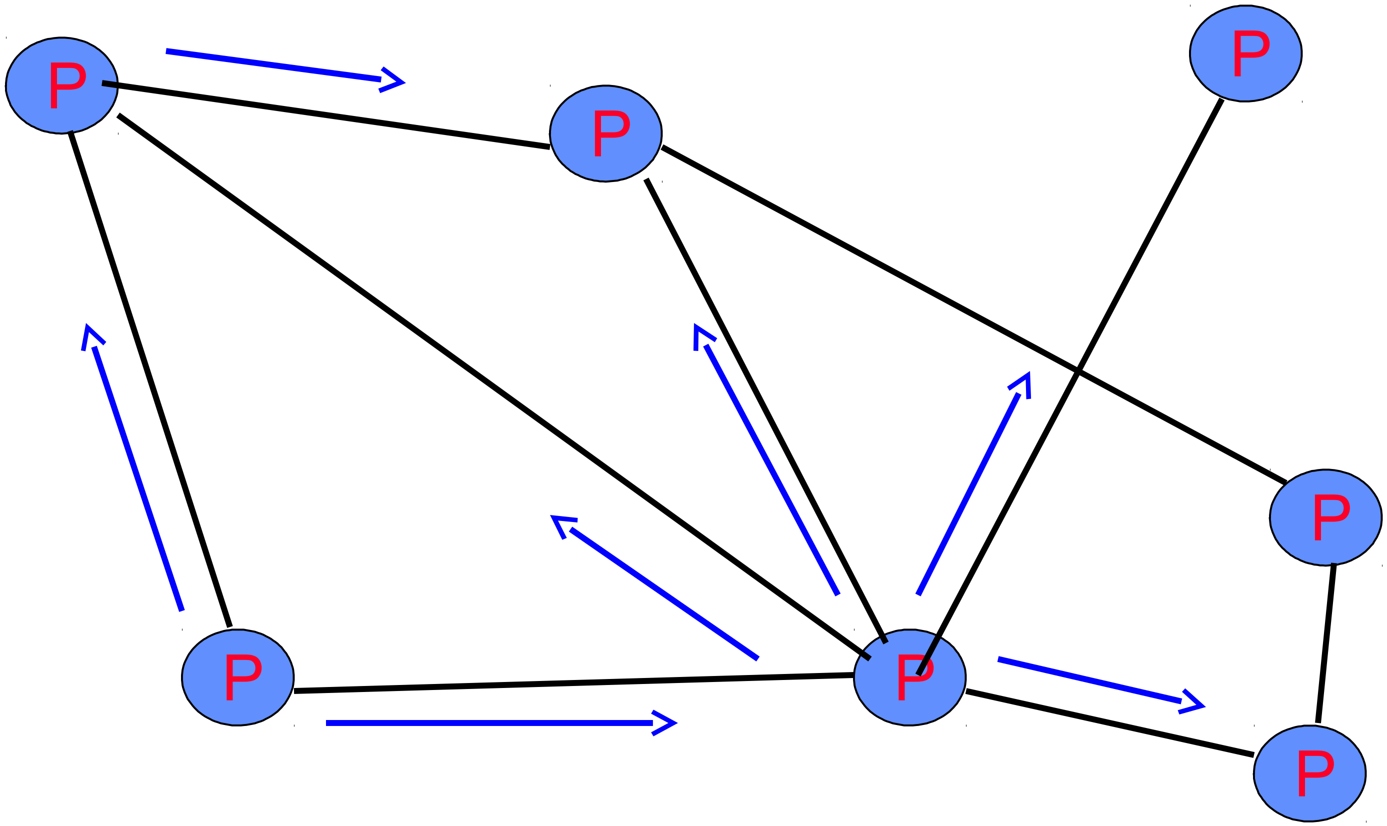
Today's Question

- How do we organize the nodes in a distributed system?
- Up to the 90s
  - 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

E.g., Gnutella



What We Don’t Want

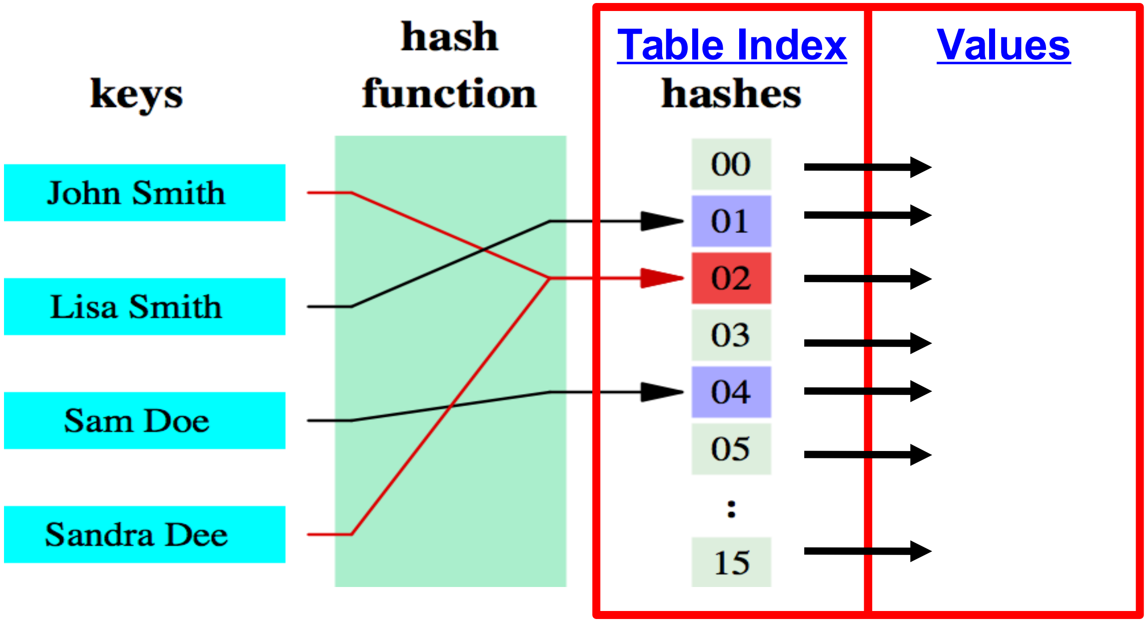
- Cost (scalability) & no guarantee for lookup

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

- 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 fast lookup-response?
- Hash table: associates keys with values



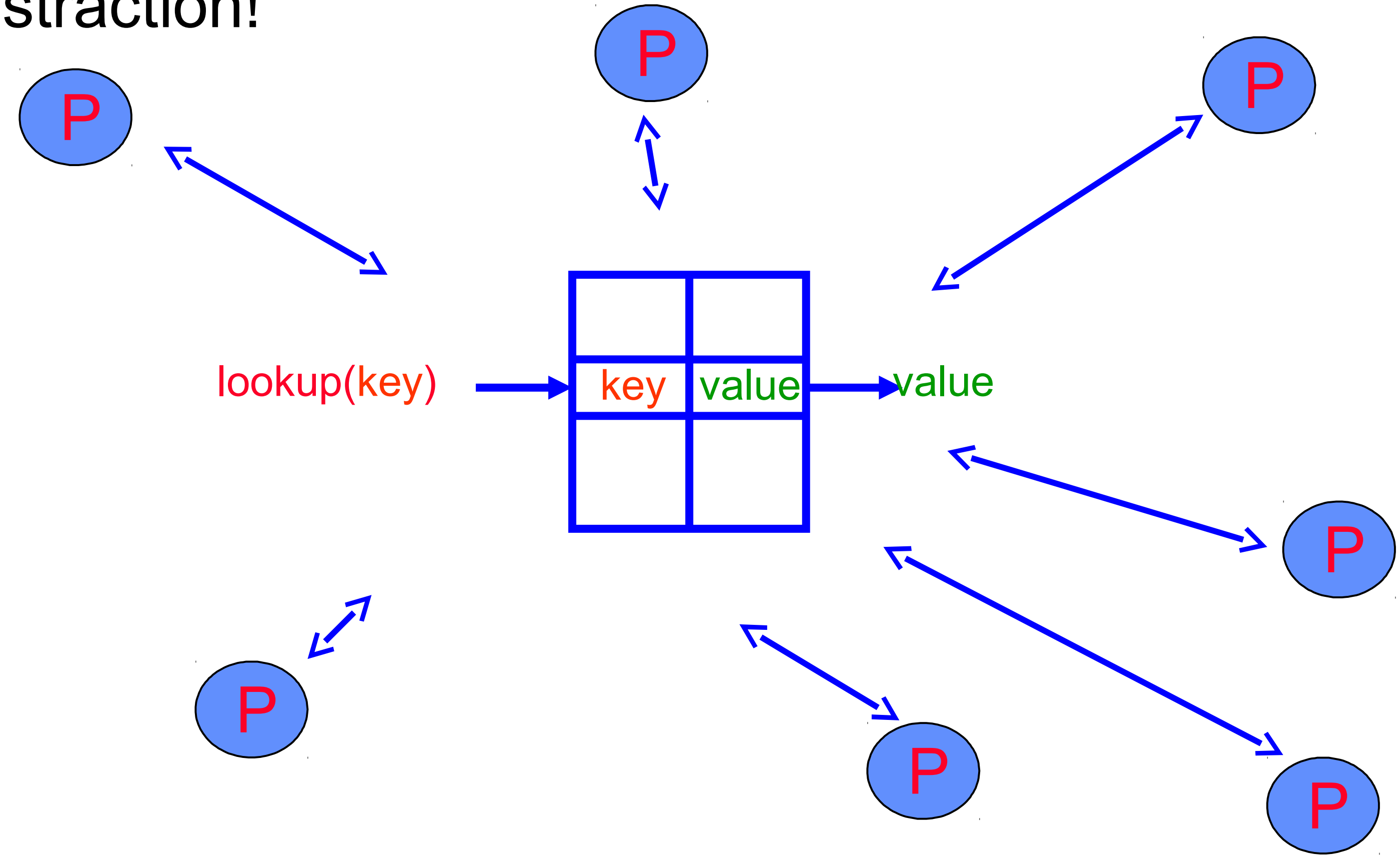
- Name-value pairs (or key-value pairs)
  - E.g., “http://www.cnn.com/foo.html” and the page contents
  - E.g., “BritneyHitMe.mp3” and “12.78.183.2”

Hashing Basics

- Hash function
  - Maps a large, possibly variable-sized datum to a small datum
  - Small datum (key) is often a single integer
  - In short: maps n-bit values into k buckets ($k << 2^n$)
  - Provides time- & space-saving data structure for lookup
- Main goals:
  - Low cost
  - Deterministic
  - Uniform distribution (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

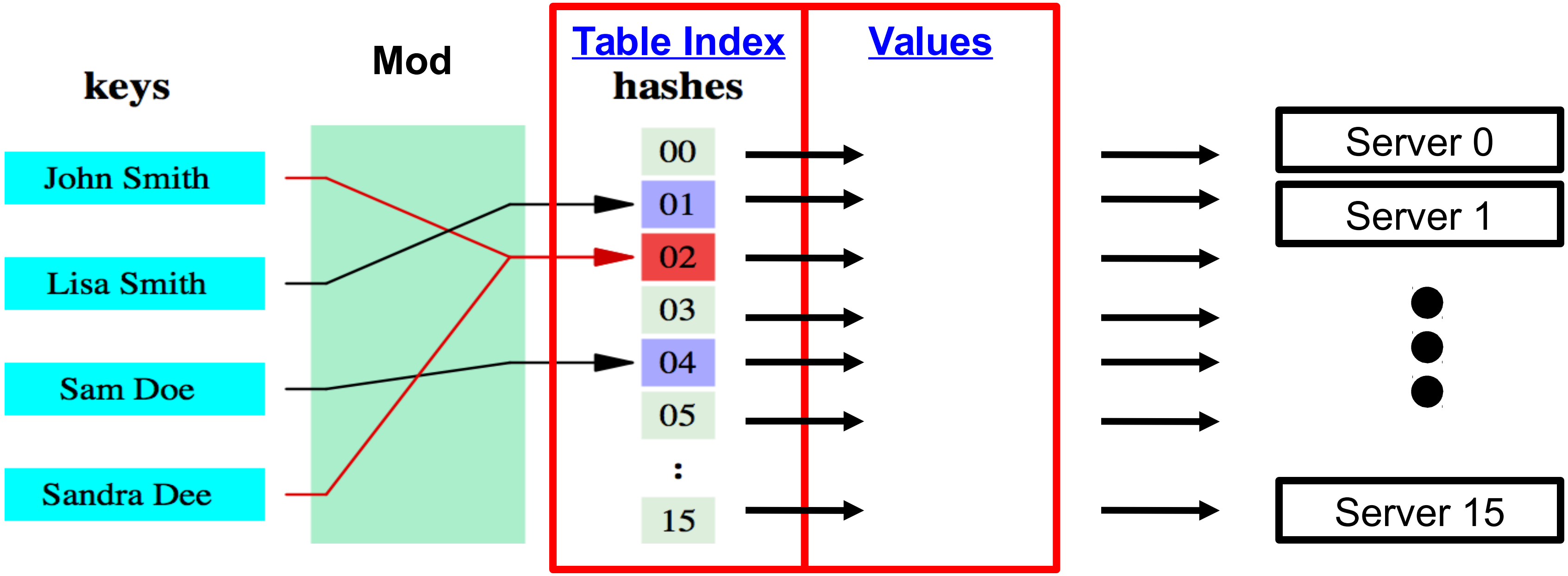
- Server-side (Napster)
- Client-local (Gnutella)
- What are the requirements (think Napster and Gnutella)?
  - Deterministic lookup
  - Low lookup time (better than linear in the system size)
  - Should balance load even with node churn
- What we'll do: partition the hash table and distribute it among 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 node churn

Where to Keep the Hash Table

- Consider the problem of data partitioning:
  - Given document X, choose one of k servers to use
- Two-level mapping
  - Hashing: Map one (or more) data item(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 when nodes leave or join)
- Consider a simple approach and its pros and cons:
  - Hashing with mod, and partitioning with buckets

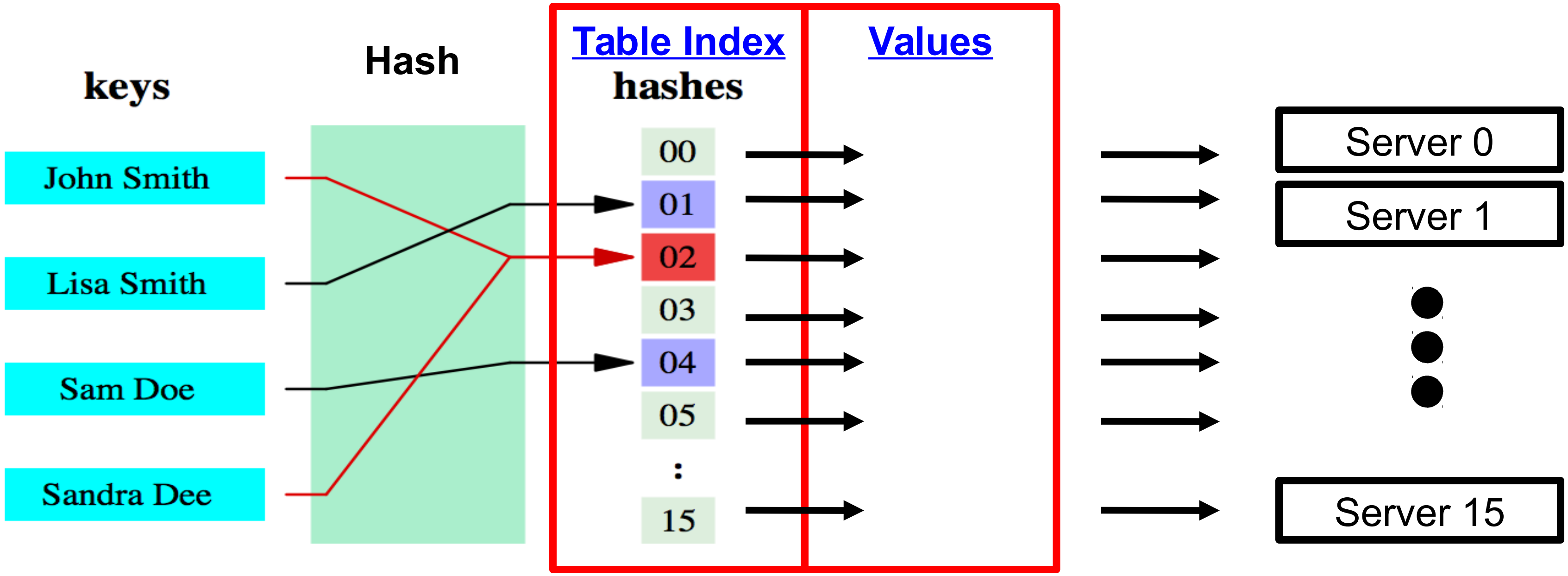
Basic Hashing and Bucket Partitioning

- 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



Basic Hashing and Bucket Partitioning

- Place X on server $i = hash(X) \bmod k$
- Problem?
  - What happens if a server fails or joins ($k \rightarrow k \pm 1$)?
  - Answer: (Almost) all entries get remapped to new nodes!

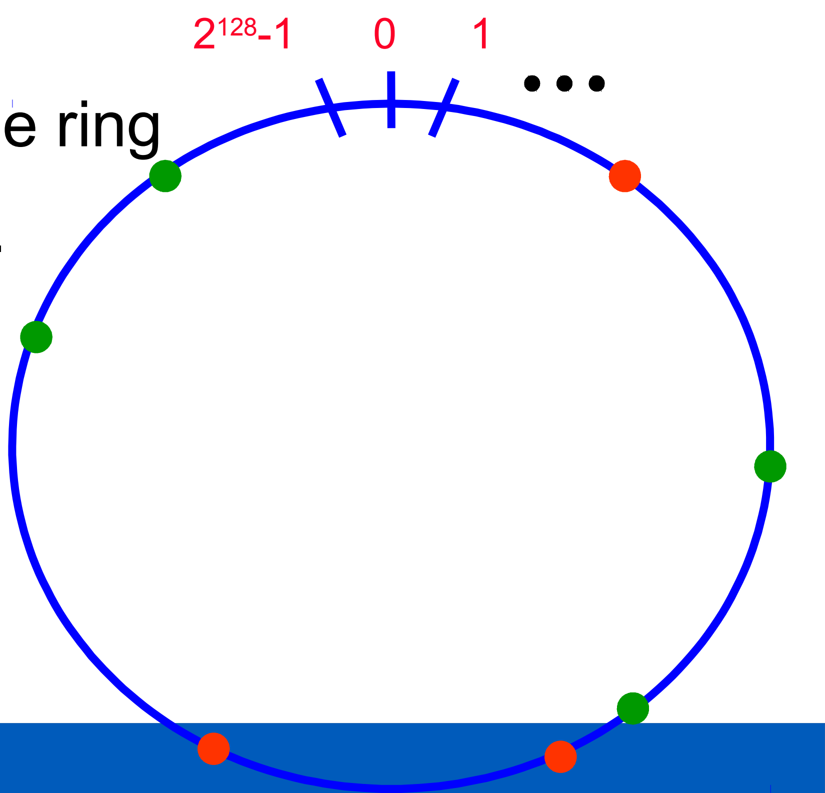


Chord DHT

- A distributed hash table system using consistent hashing
- Organizes nodes in a ring
- Maintains neighbors for correctness and shortcuts for performance
- DHT in general
  - Structured peer-to-peer (as opposed to Napster, Gnutella, *etc.*)
  - Used as a foundation for other systems
    - "Trackerless" BitTorrent clients
    - Amazon Dynamo
    - Distributed filesystems
    - *etc.*
- Demonstrates 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.

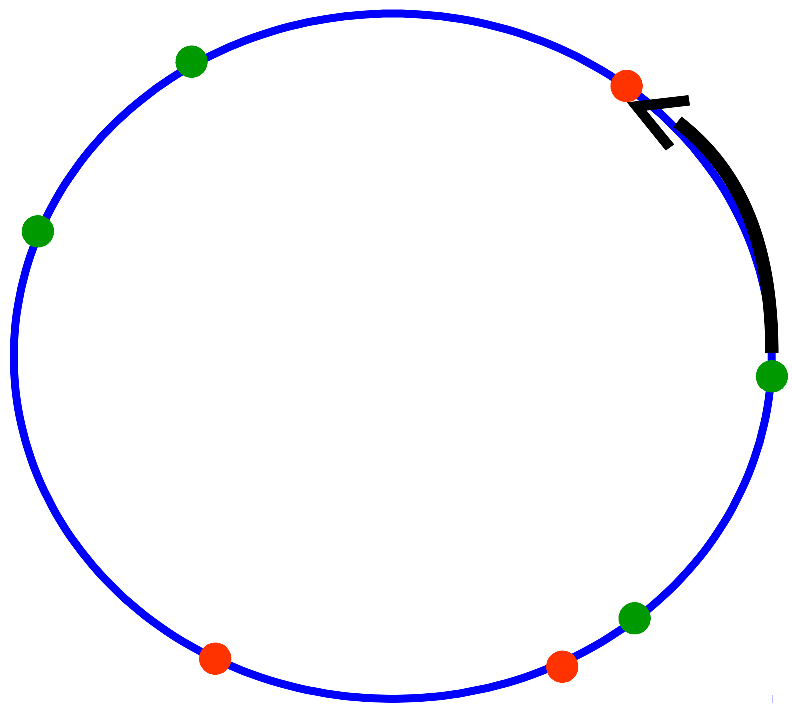


ID space forms a ring

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

Chord: Consistent Hashing

- Partitioning: Data item is mapped to its "successor" node
- Advantages
  - Even distribution
  - Few changes as nodes come and go…



Hash (name) → object_id

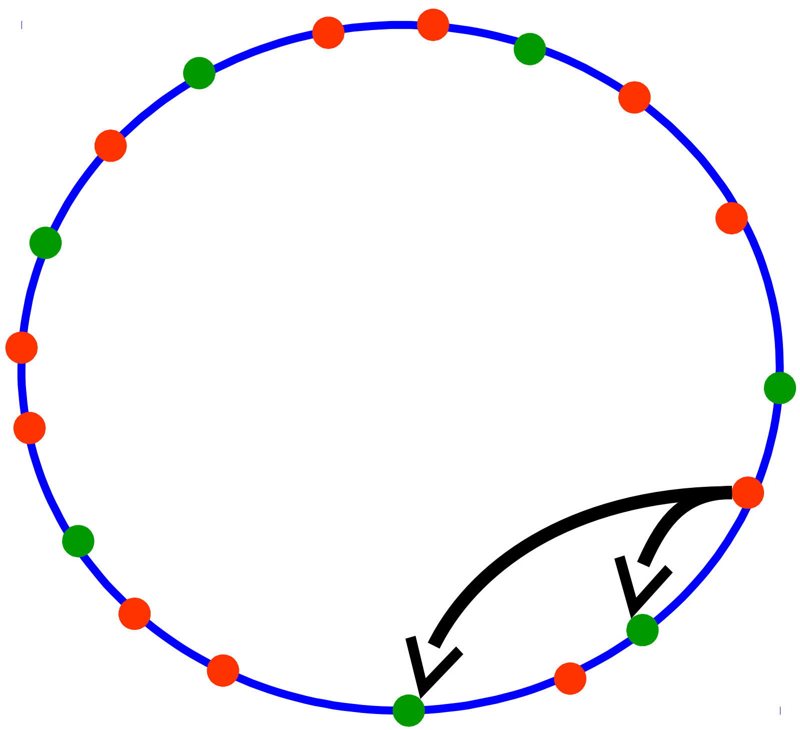
Hash(IP_address) → node_id

Chord: When nodes come and go…

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

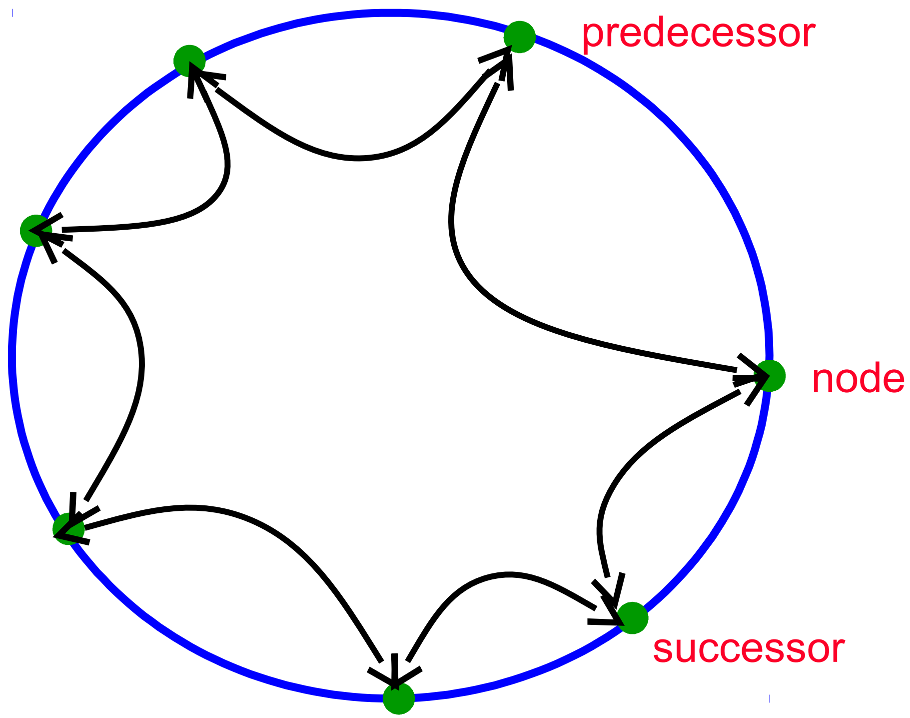
Hash (name) → object_id

Hash(IP_address) → node_id



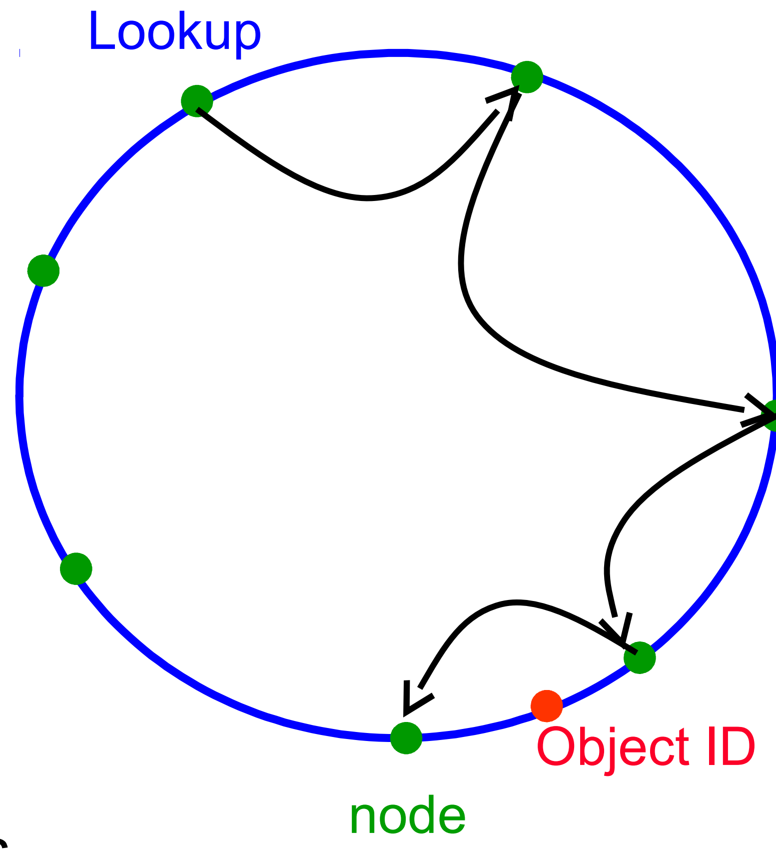
Chord: Node Organization

- Maintain a circularly linked list around the ring
  - Every node has a predecessor and successor
- Separate join and leave protocols



Chord: Basic Lookup

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



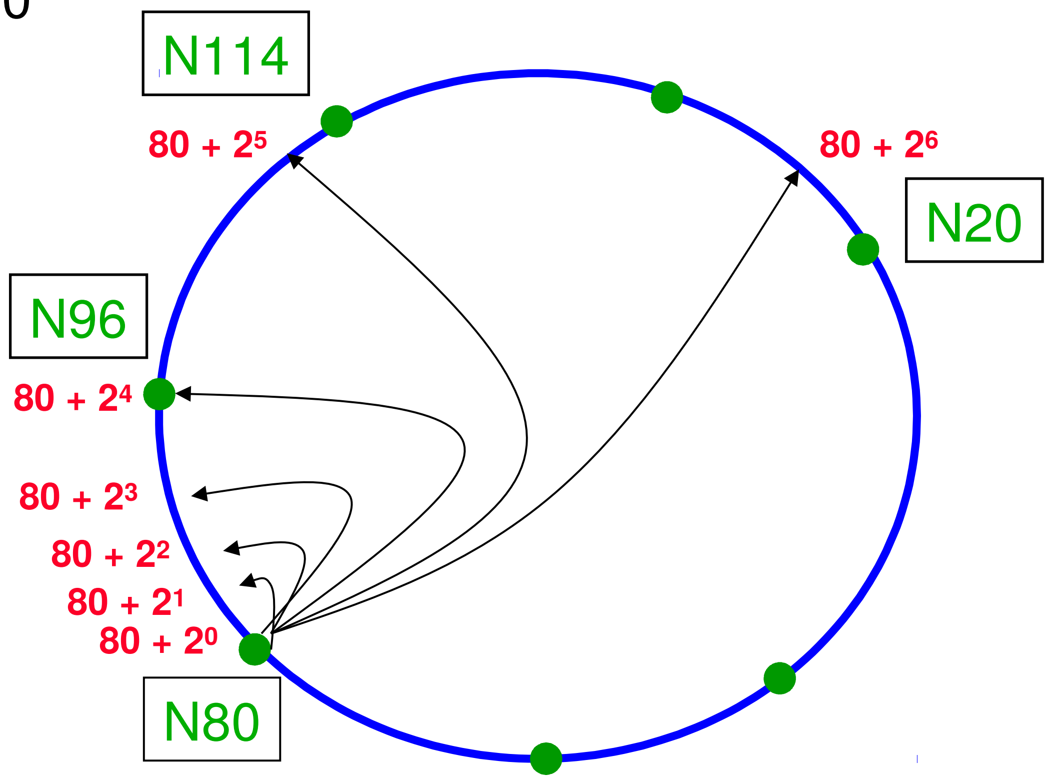
- Route hop by hop via successors
  - O(n) hops to find destination id

Chord: Efficient Lookup — Fingers

- i^{th} entry at peer with id n is first peer with:
  - id >= $n + 2^i (\text{mod} 2^m)$

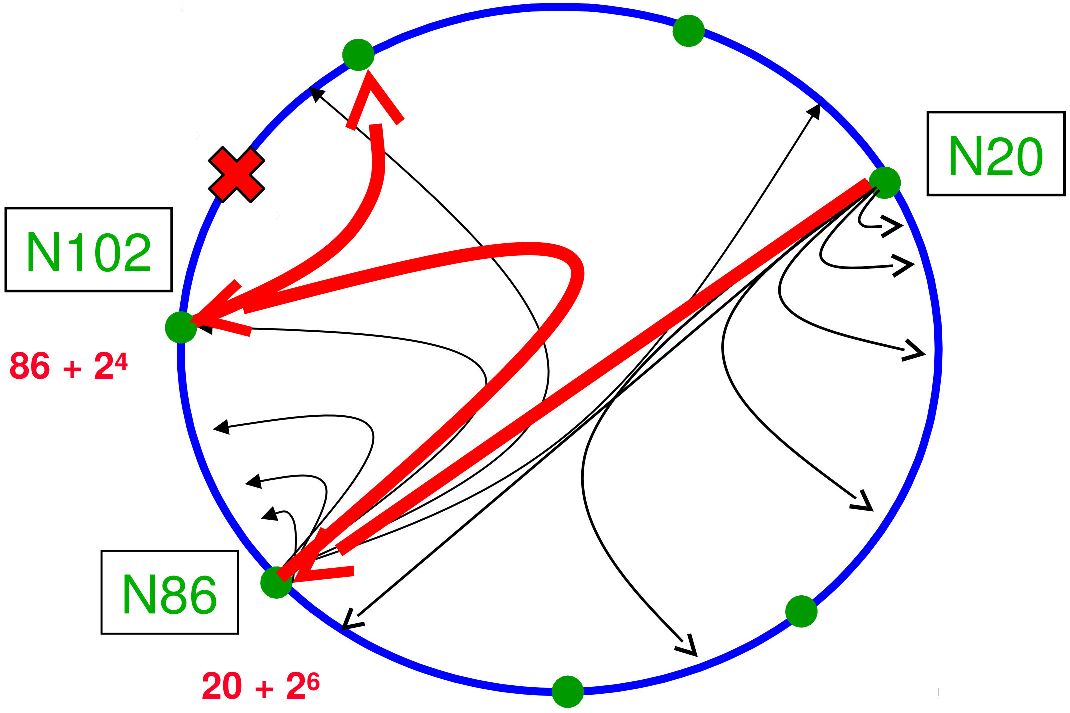
Finger Table at N80

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 a neighbor/peer doesn't respond for some time, find a new one
- When a node leaves
  - 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
  - Provides a hash table abstraction
  - Partitions the hash table and distributes partitions over the nodes
  - Uses peer-to-peer structure
- Chord DHT
  - Based on consistent hashing
  - Balances hash table partitions over the nodes
  - Basic lookup based on successors
  - Efficient lookup through fingers

References

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* Proceedings of ACM SIGCOMM. August 2001. **Required Reading.** https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

Acknowledgements

- These slides are by Steve Ko, used with permission (and lightly modified) by Ethan Blanton.
- These slides contain material developed and copyrighted by Indranil Gupta (UIUC), Michael Freedman (Princeton), and Jennifer Rexford (Princeton).