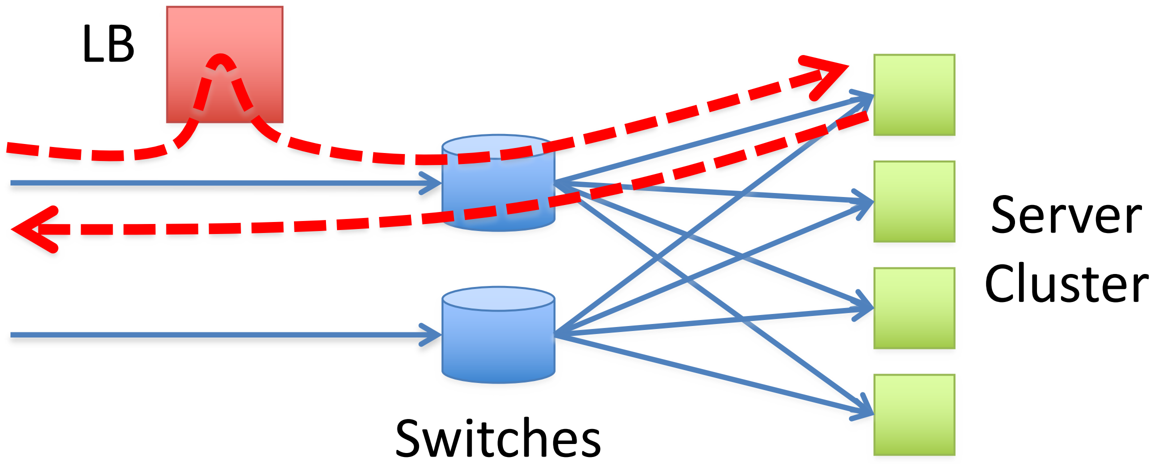
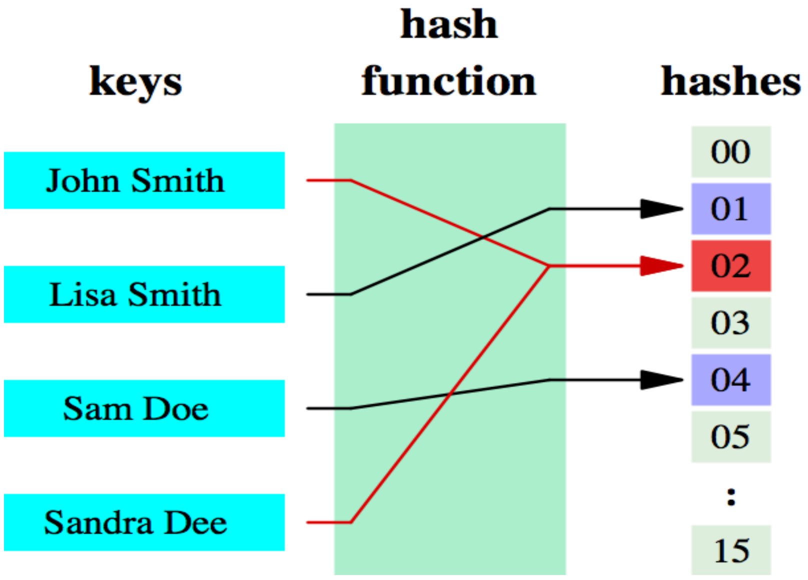


Hashing in Networked Systems

Note: The slides are adapted from the materials from Prof. Richard Han at CU Boulder and Profs. Jennifer Rexford and Mike Freedman at Princeton University, and the networking book (Computer Networking: A Top Down Approach) from Kurose and Ross.

Hashing

- 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)

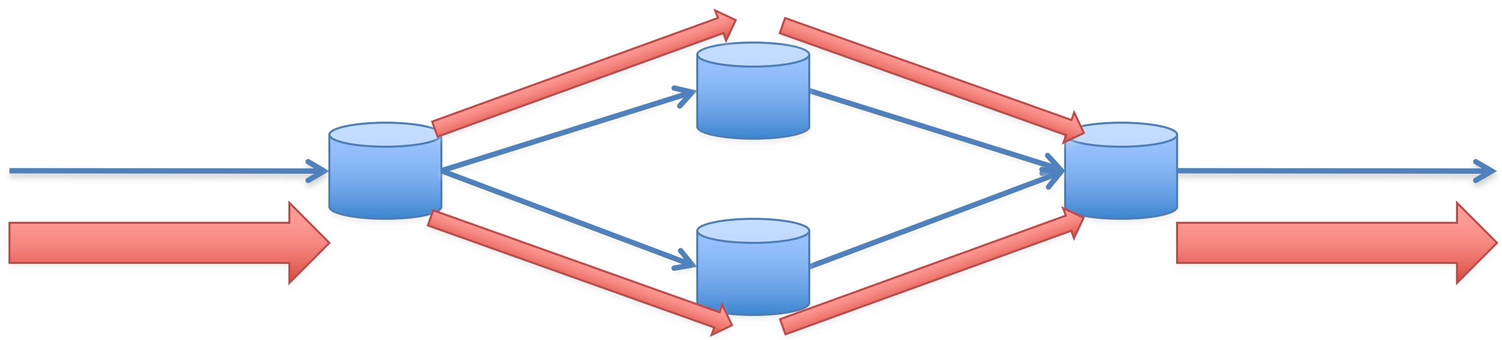


Today's outline

- Uses of hashing
  - Equal-cost multipath routing in switches
  - Network load balancing in server clusters
  - Per-flow statistics in switches (QoS, IDS)
  - Caching in cooperative CDNs and P2P file sharing
  - Data partitioning in distributed storage services
- Various hashing strategies
  - Modulo hashing
  - Consistent hashing
  - Bloom Filters

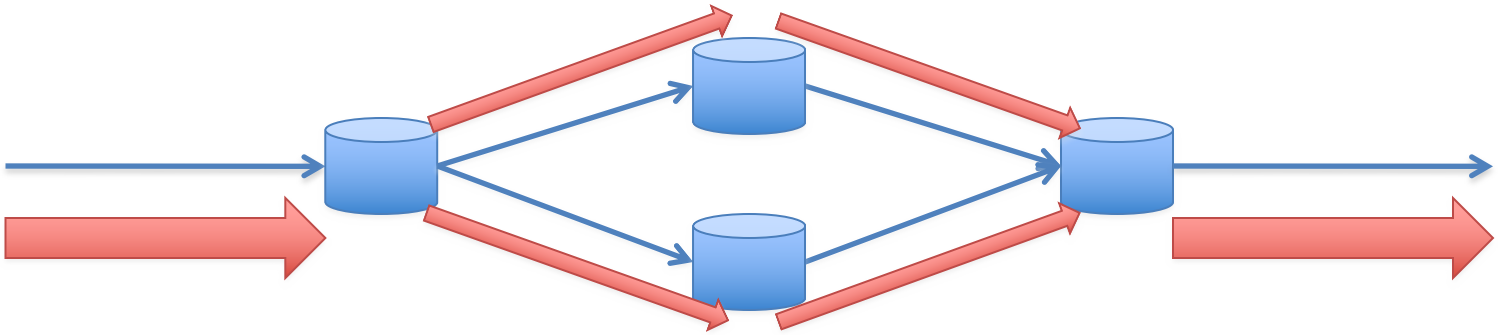
Uses of Hashing

Equal-cost multipath routing (ECMP)



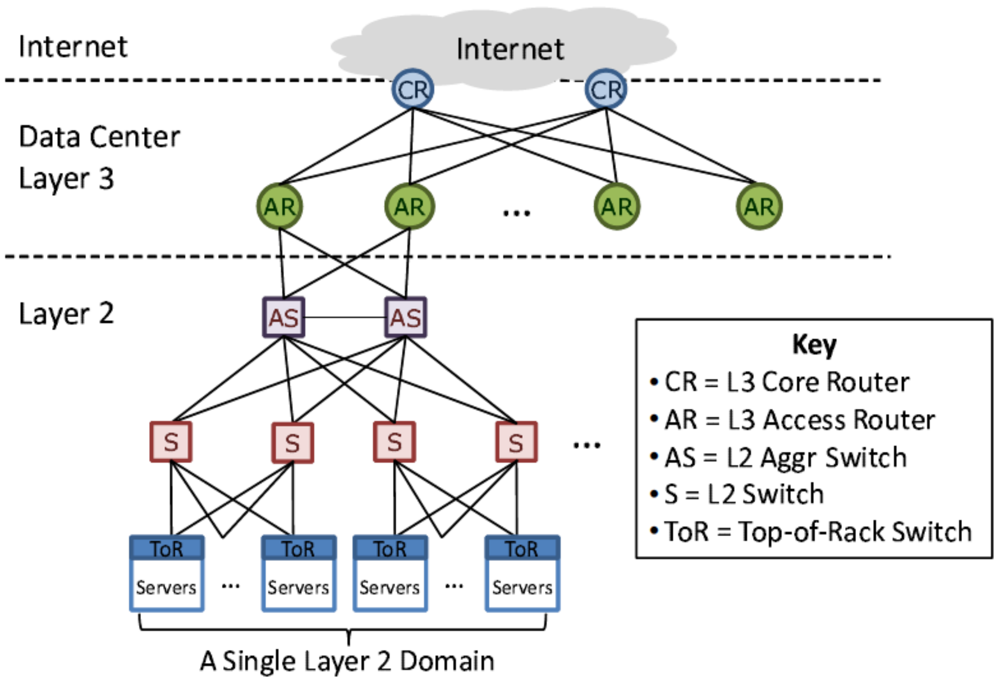
- ECMP
  - Multipath routing strategy that splits traffic over multiple paths for load balancing
- Why not just round-robin packets?
  - Reordering (lead to triple duplicate ACK in TCP?)
  - Different RTT per path (for TCP RTO)…
  - Different MTUs per path

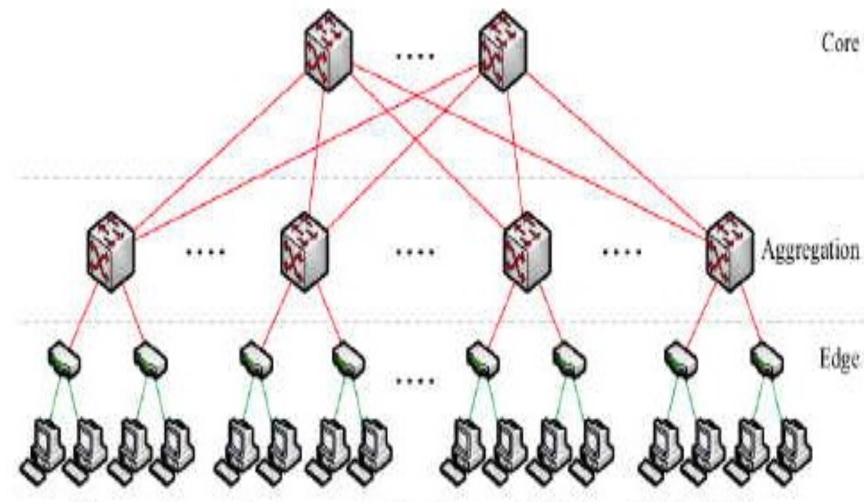
Equal-cost multipath routing (ECMP)



- Path-selection via hashing
  - # buckets = # outgoing links
  - Hash network information (source/dest IP addrs) to select outgoing link: preserves flow affinity

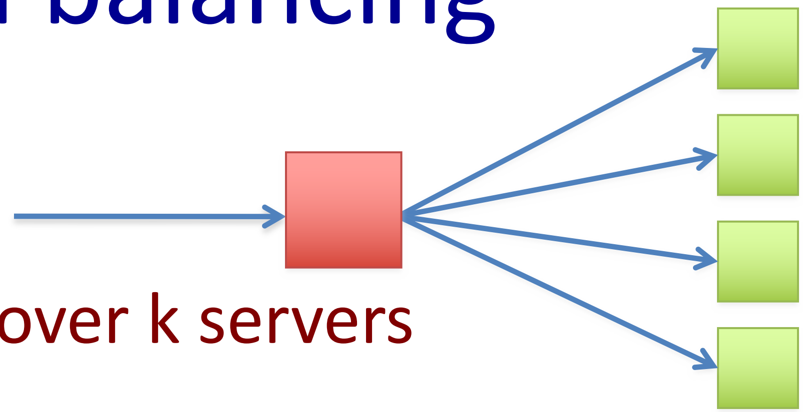
Now: ECMP in datacenters





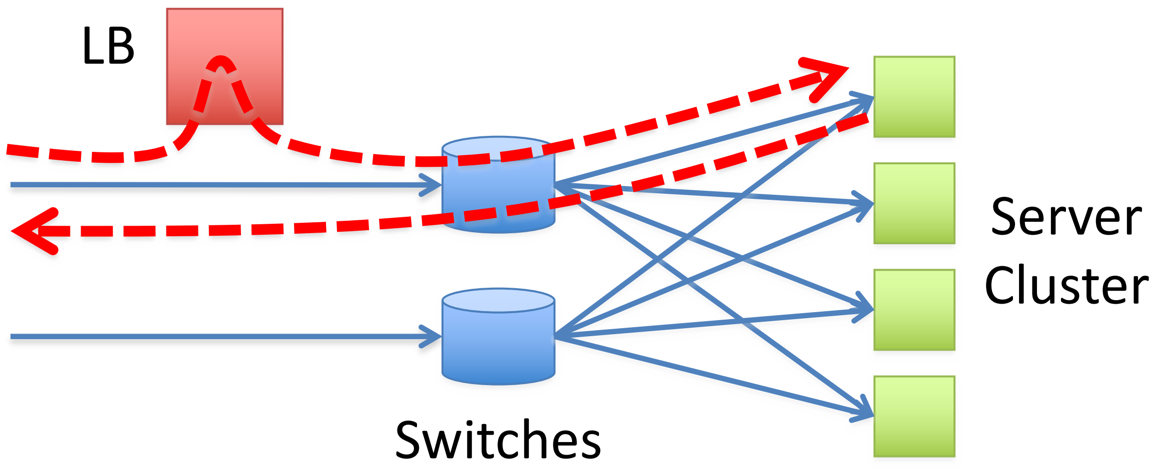
- Datacenter networks are multi-rooted tree
  - Goal: Support for 100,000s of servers
  - Recall Ethernet spanning tree problems: No loops
  - L3 routing and ECMP: Take advantage of multiple paths

Network load balancing



- Goal: Split requests evenly over k servers
  - Map new flows to any server
  - Packets of existing flows continue to use same server
- 3 approaches
  - Load balancer terminates TCP, opens own connection to server
  - Virtual IP / Dedicated IP (VIP/DIP) approaches
    - One global-facing virtual IP represents all servers in cluster
    - Hash client's network information (source IP:port)
    - NAT approach: Replace virtual IP with server's actual IP
    - Direct Server Return (DSR)

Load balancing with DSR



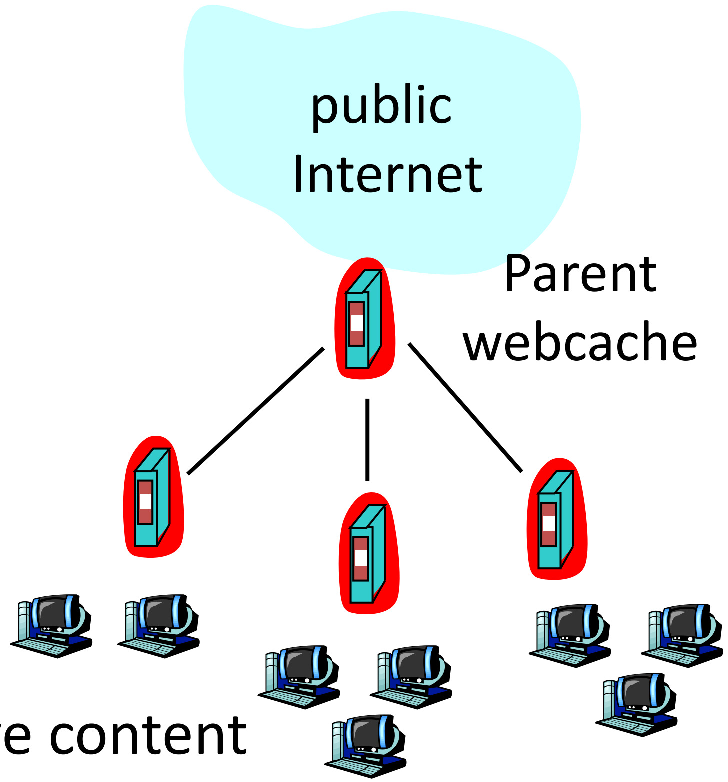
- Servers bind to both virtual and dedicated IP
- Load balancer just replaces dest MAC addr
- Server sees client IP, responds directly
  - Packet in reverse direction do not pass through load balancer
  - Greater scalability, particularly for traffic with assymmetric bandwidth (e.g., HTTP GETs)

Per-flow state in switches

- Switches often need to maintain connection records or per-flow state
  - Quality-of-service for flows
  - Flow-based measurement and monitoring
  - Payload analysis in Intrusion Detection Systems (IDSs)
- On packet receipt:
  - Hash flow information (packet 5-tuple)
  - Perform lookup if packet belongs to known flow
  - Otherwise, possibly create new flow entry
  - Probabilistic match (false positives) may be okay

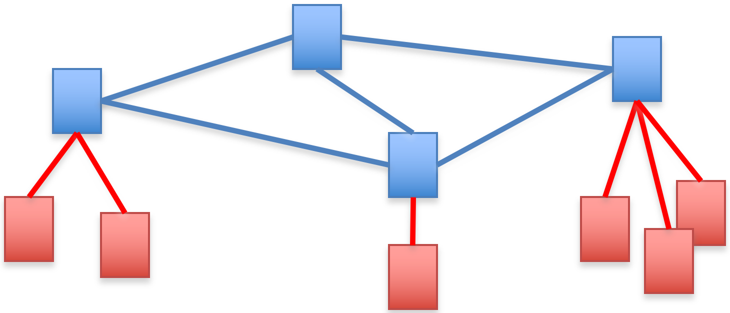
Cooperative Web CDNs

- Tree-like topology of cooperative web caches
  - Check local
  - If miss, check siblings / parent



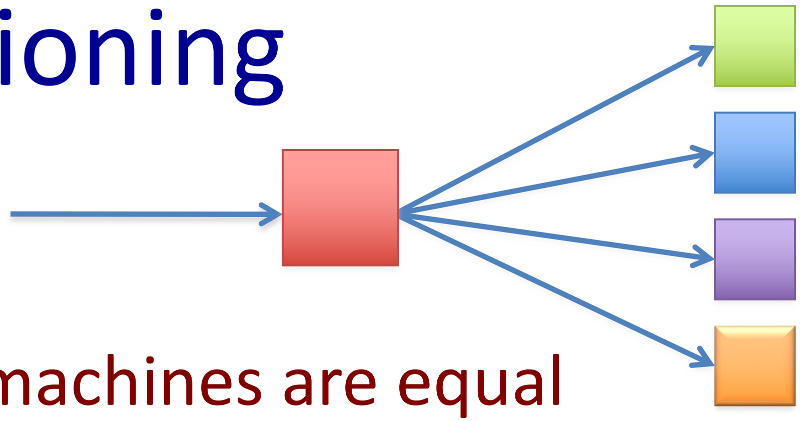
- One approach
  - Internet Cache Protocol (ICP)
  - UDP-based lookup, short timeout
- Alternative approach
  - A priori guess is siblings/children have content
  - Nodes share hash table of cached content with parent / siblings
  - Probabilistic check (false positives) okay, as actual ICP lookup to neighbor could just return false

Hash tables in P2P file-sharing



- Two-layer network (e.g., Gnutella, Kazaa)
  - Ultrapeers are more stable, not NATted, higher bandwidth
  - Leaf nodes connect with 1 or more ultrapeers
- Ultrapeers handle content searchers
  - Leaf nodes send hash table of content to ultrapeers
  - Search requests flooded through ultrapeer network
  - When ultrapeer gets request, checks hash tables of its children for match

Data partitioning



- Network load balancing: All machines are equal
- Data partitioning: Machines store different content

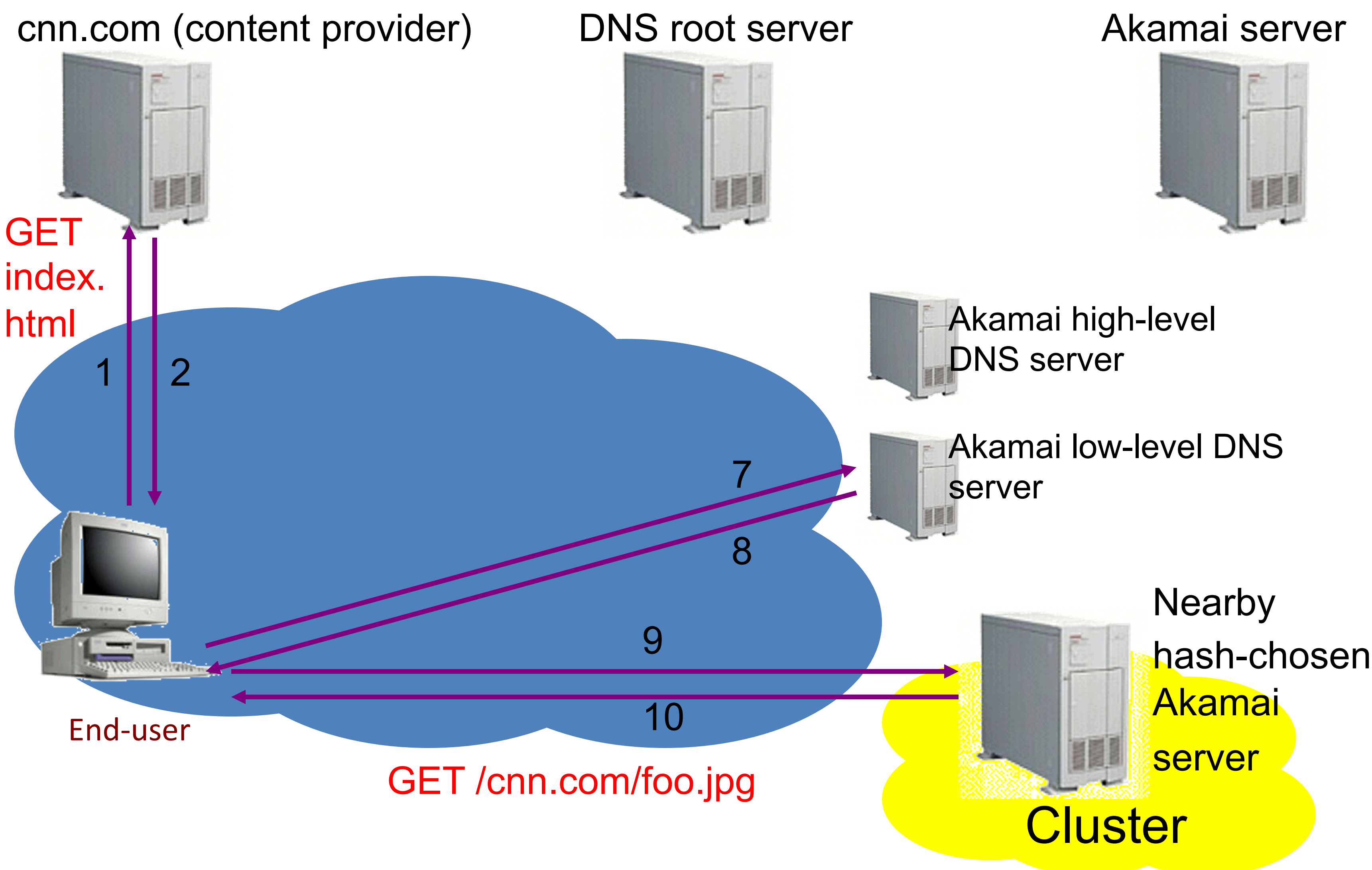
- Non-hash-based solution
  - "Directory" server maintains mapping from O(entries) to machines (e.g., Network file system, Google File System)
  - Named data can be placed on any machine

- Hash-based solution
  - Nodes maintain mappings from O(buckets) to machines
  - Data placed on the machine that owns the name's bucket

Examples of data partitioning

- Akamai
  - 1000 clusters around Internet, each >= 1 servers
  - Hash (URL's domain) to map to one server
  - Akamai DNS aware of hash function, returns machine that
    1. is in geographically-nearby cluster
    2. manages particular customer domain
- Memcached (Facebook, Twitter, …)
  - Employ k machines for in-memory key-value caching
  - On read:
    - Check memcache
    - If miss, read data from DB, write to memcache
  - On write: invalidate cache, write data to DB

How Akamai Works – Already Cached



Hashing Techniques

Basic Hash Techniques

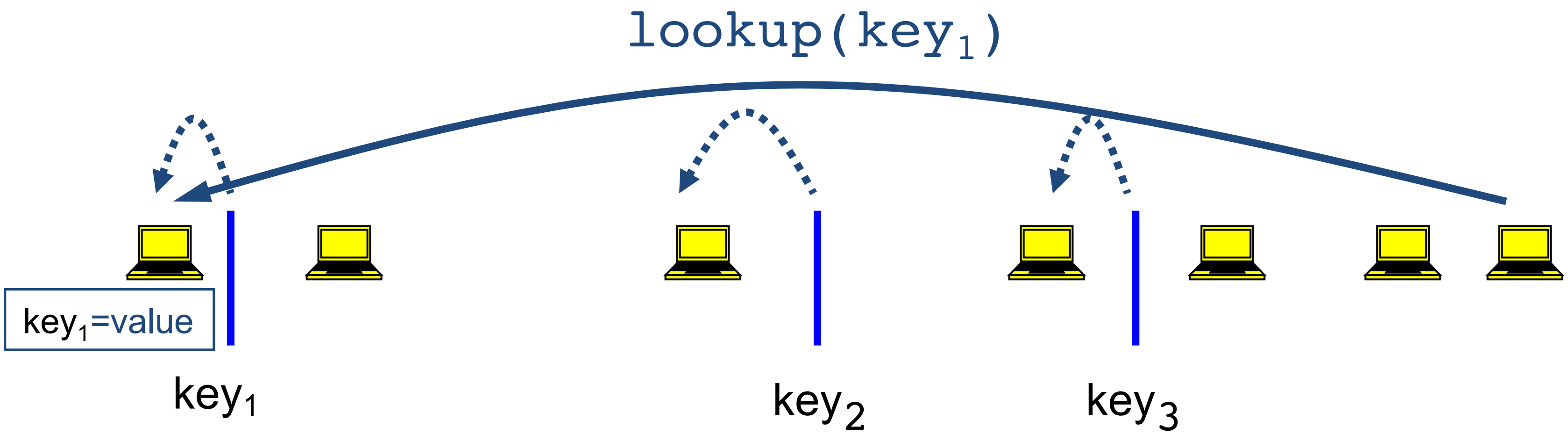
- Simple approach for uniform data
  - If data distributed uniformly over N, for N >> n
  - Hash fn = <data> mod n
  - Fails goal of uniformity if data not uniform

- Non-uniform data, variable-length strings
  - Typically split strings into blocks
  - Perform rolling computation over blocks
    - CRC32 checksum
    - Cryptographic hash functions (SHA-1 has 64 byte blocks)

Applying Basic Hashing

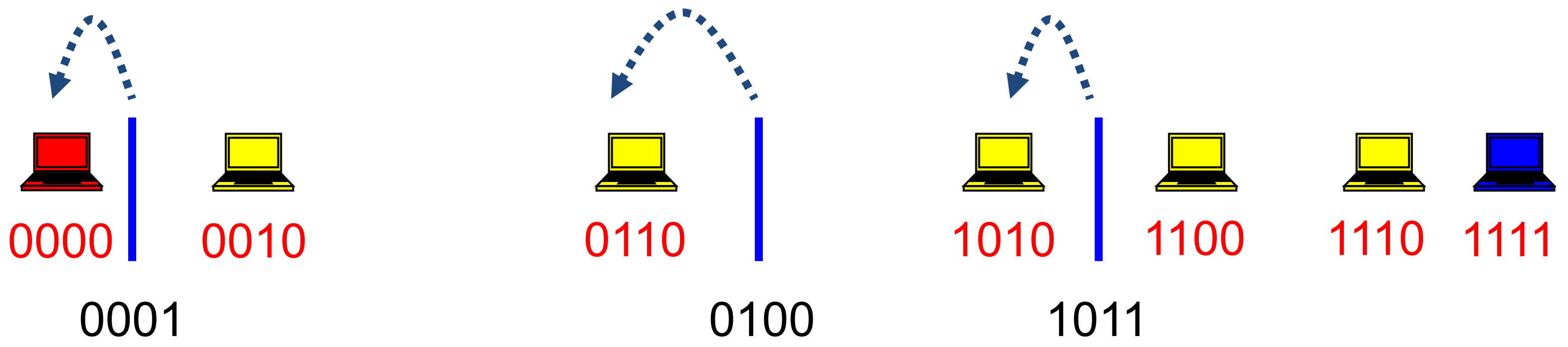
- Consider problem of data partition:
  - Given document X, choose one of k servers to use
- Suppose we use modulo hashing
  - Number servers 1..k
  - Place X on server *i = (X mod k)*
    - Problem? Data may not be uniformly distributed
  - Place X on server *i = hash (X) mod k*
    - Problem?
      - What happens if a server fails or joins ($k \rightarrow k \pm 1$)?
      - What is different clients has different estimate of k?
      - Answer: All entries get remapped to new nodes!

Consistent Hashing



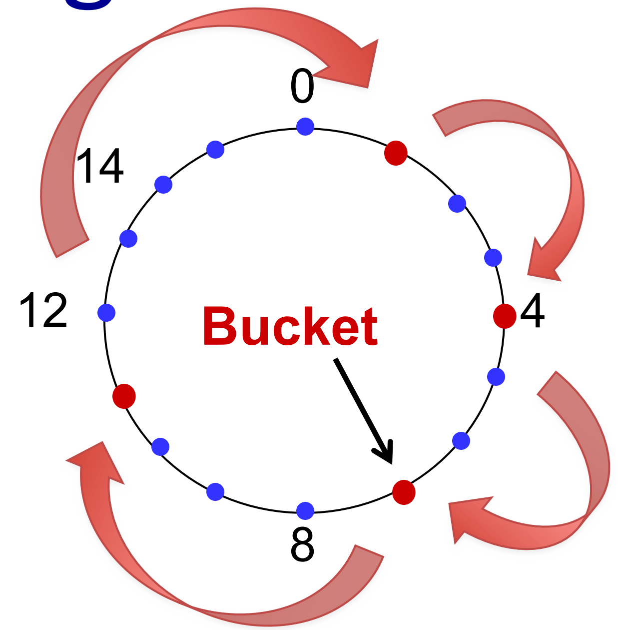
- Consistent hashing partitions key-space among nodes
- Contact appropriate node to lookup/store key
  - Blue node determines red node is responsible for key_1
  - Blue node sends lookup or insert to red node

Consistent Hashing



- Partitioning key-space among nodes
  - Nodes choose random identifiers: e.g., hash(IP)
  - Keys randomly distributed in ID-space: e.g., hash(URL)
  - Keys assigned to node “nearest” in ID-space
  - Spreads ownership of keys evenly across nodes

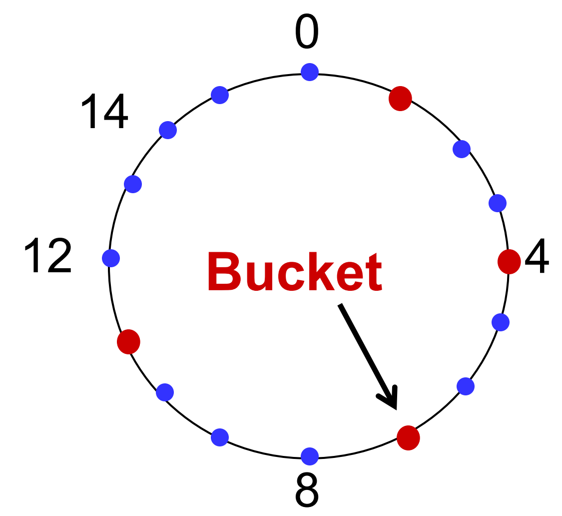
Consistent Hashing



- Construction
  - Assign n hash buckets to random points on mod 2^k circle; hash key size = k
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket
    - *successor* (key) → bucket
- Desired features
  - Balanced: No bucket has disproportionate number of objects
  - Smoothness: Addition/removal of bucket does not cause movement among existing buckets (only immediate buckets)
  - Spread and load: Small set of buckets that lie near object

Consistent hashing and failures

- Consider network of n nodes
- If each node has 1 bucket
  - Owns $1/n^{th}$ of keyspace *in expectation*



- If a node fails:

(A) Nobody owns keyspace (B) Keyspace assigned to random node

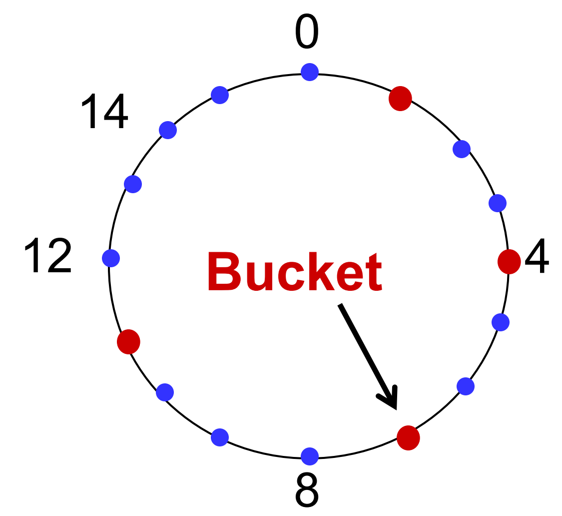
(C) Successor owns keyspaces (D) Predecessor owns keyspace

- After a node fails:

(A)Load is equally balanced over all nodes

(B)Some node has disproportional load compared to others

Consistent hashing and failures



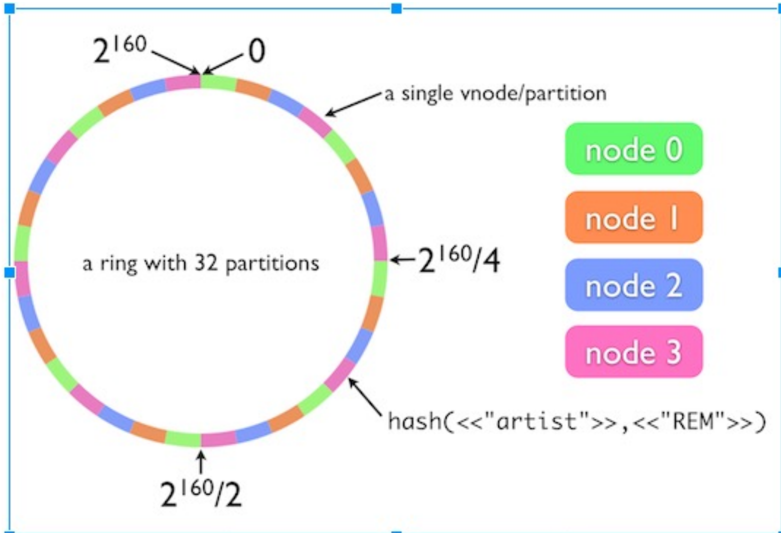
- Consider network of n nodes
- If each node has 1 bucket
  - Owns $1/n^{th}$ of keyspace *in expectation*

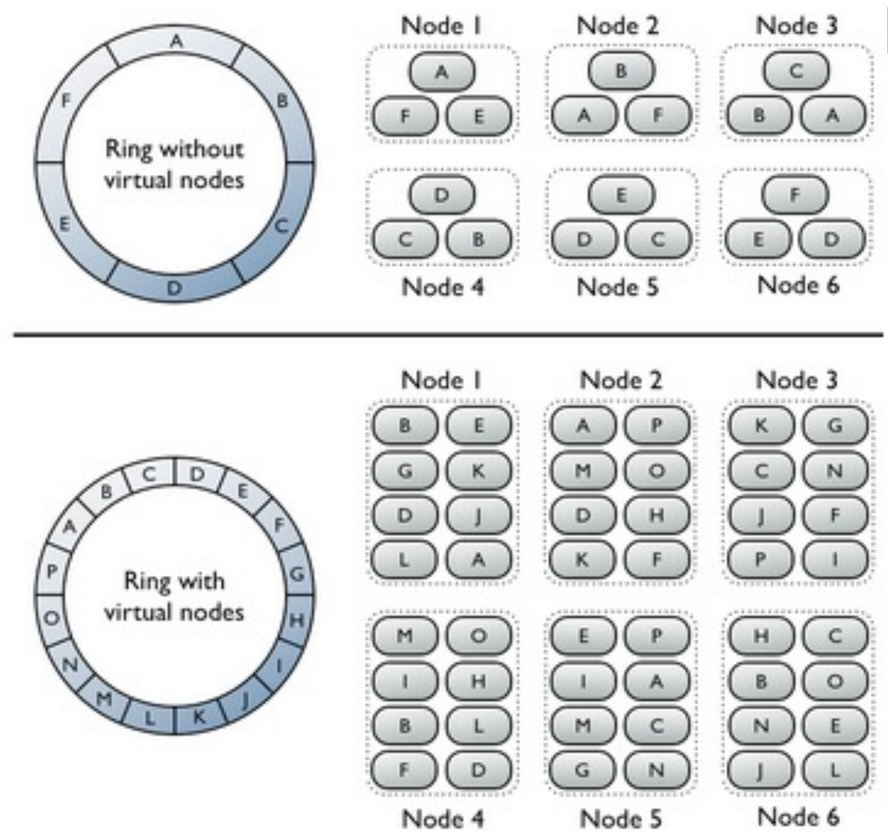
- If a node fails:
  - Its *successor* takes over bucket
  - Achieves smoothness goal: Only localized shift, not O(n)
  - But now successor owns 2 buckets: keyspace of size *2/n*

- Instead, if each node maintains *v* random nodeIDs, not 1
  - "Virtual" nodes spread over ID space, each of size *1 / vn*
  - Upon failure, v successors take over, each now stores *(v+1) / vn*

Example: Cassandra

- Cassandra adopts consistent hashing with virtual nodes for data partitioning



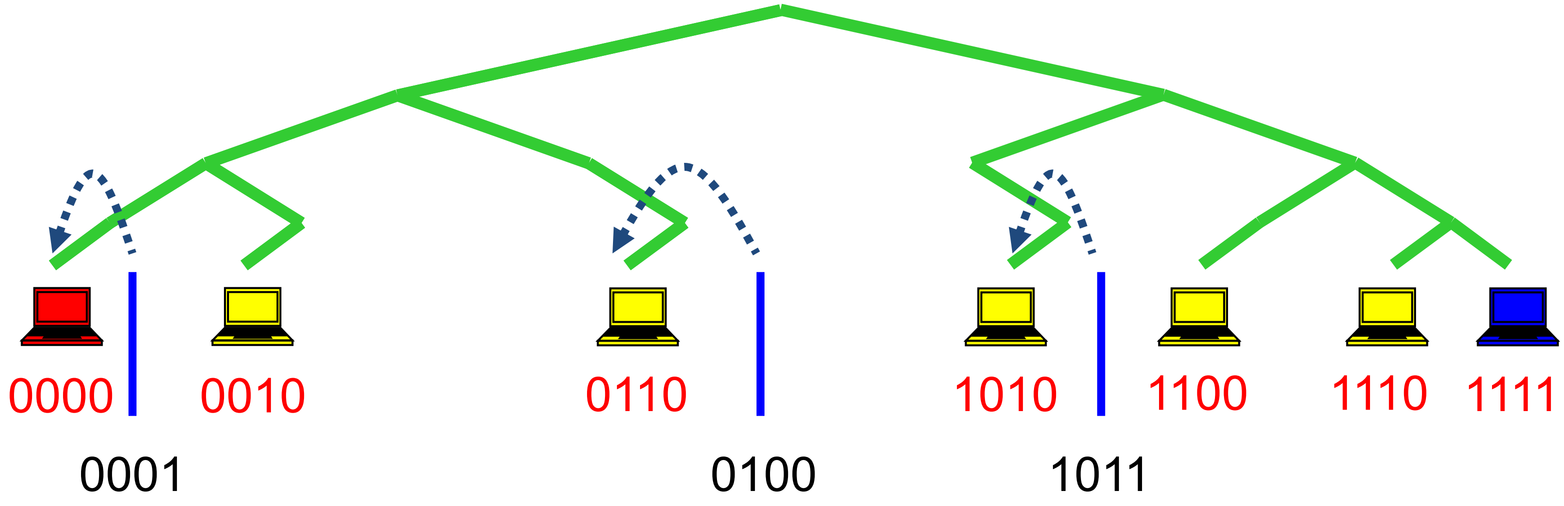


[Credit: https://blog.imaginea.com/consistent-hashing-in-cassandra/](https://blog.imaginea.com/consistent-hashing-in-cassandra/)

Consistent hashing vs. DHTs

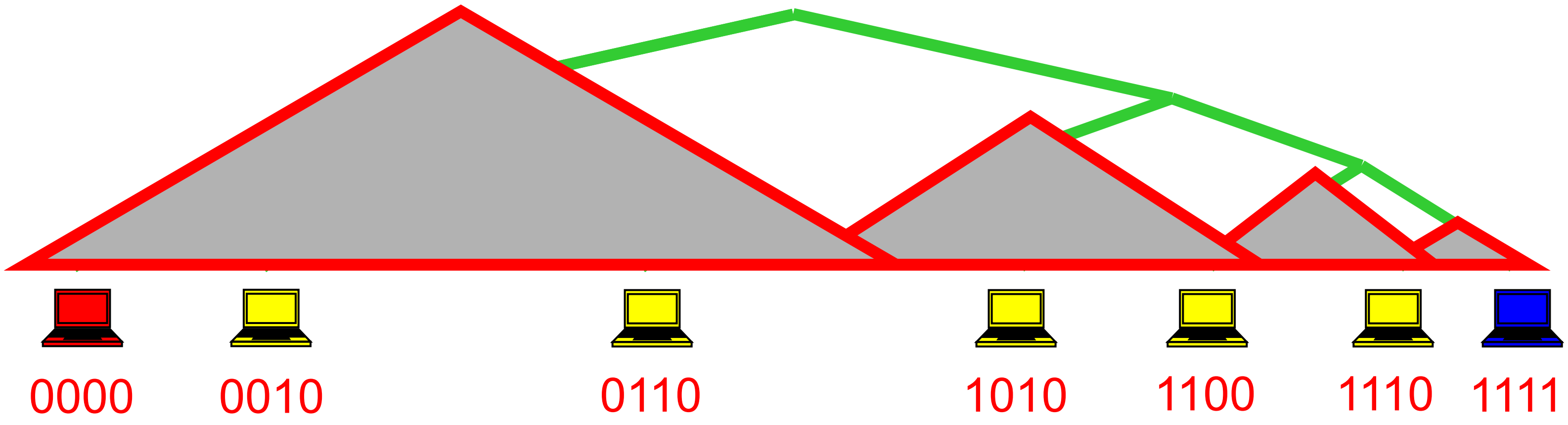
	Consistent Hashing	Distributed Hash Tables
Routing table size	O(n)	O(log n)
Lookup / Routing	O(1)	O(log n)
Join/leave: Routing updates	O(n)	O(log n)
Join/leave: Key Movement	O(1)	O(1)

Distributed Hash Table



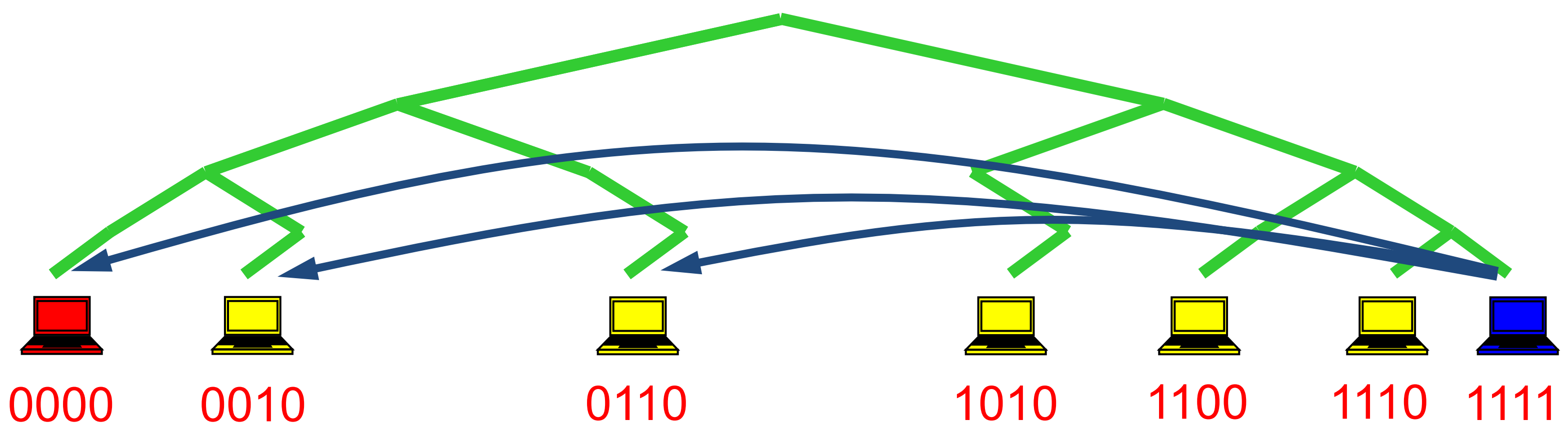
- Nodes' neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node

Distributed Hash Table



- Nodes' neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node
  - At least one neighbor known per subtree of increasing size /distance from node

Distributed Hash Table

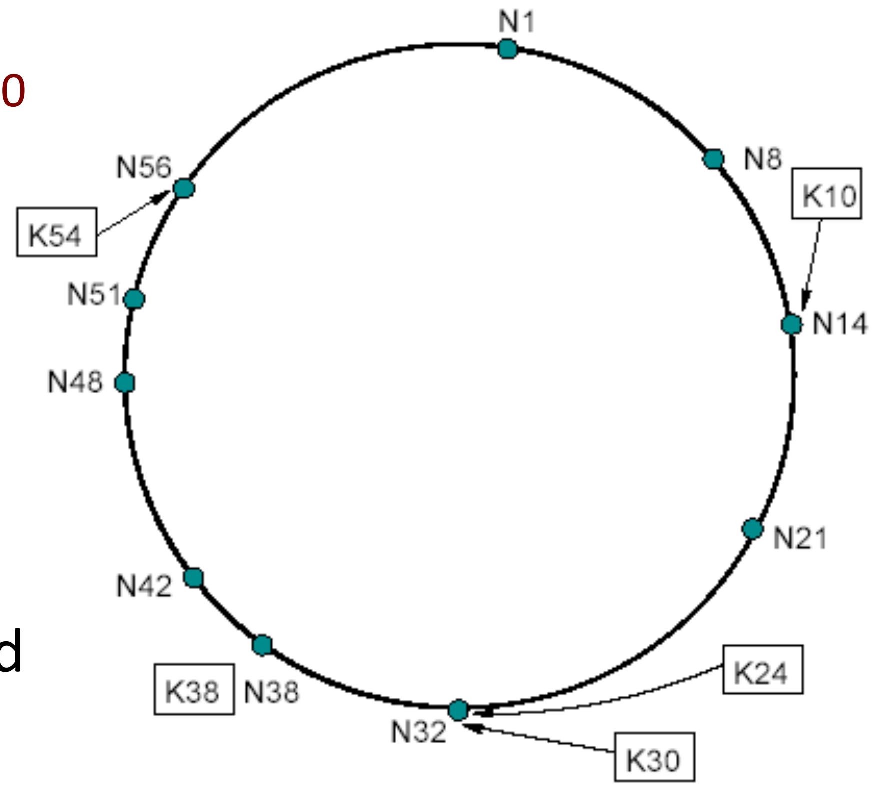


- Nodes’ neighbors selected from particular distribution
  - Visual keyspace as a tree in distance from a node
  - At least one neighbor known per subtree of increasing size /distance from node
- Route greedily towards desired key via overlay hops

The Chord DHT

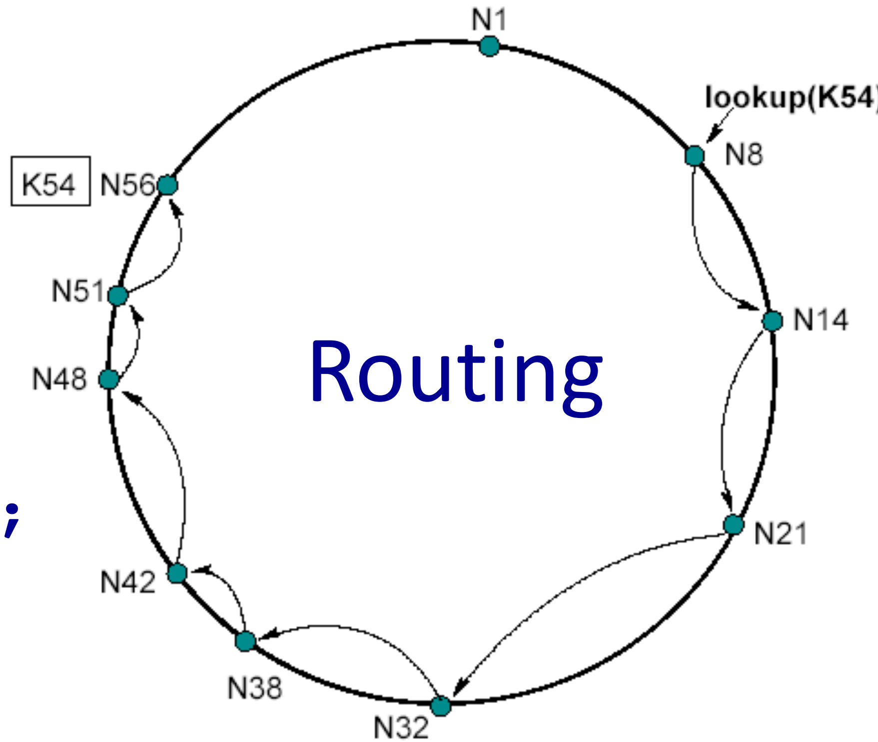
- Chord ring: ID space mod 2^{160}
  - *nodeid = SHA1 (IP address, i)*

    for i=1..v virtual IDs
  - *keyid = SHA1 (name)*
- Routing correctness:
  - Each node knows successor and predecessor on ring
- Routing efficiency:
  - Each node knows O(log n) well-distributed neighbors



Basic lookup in Chord

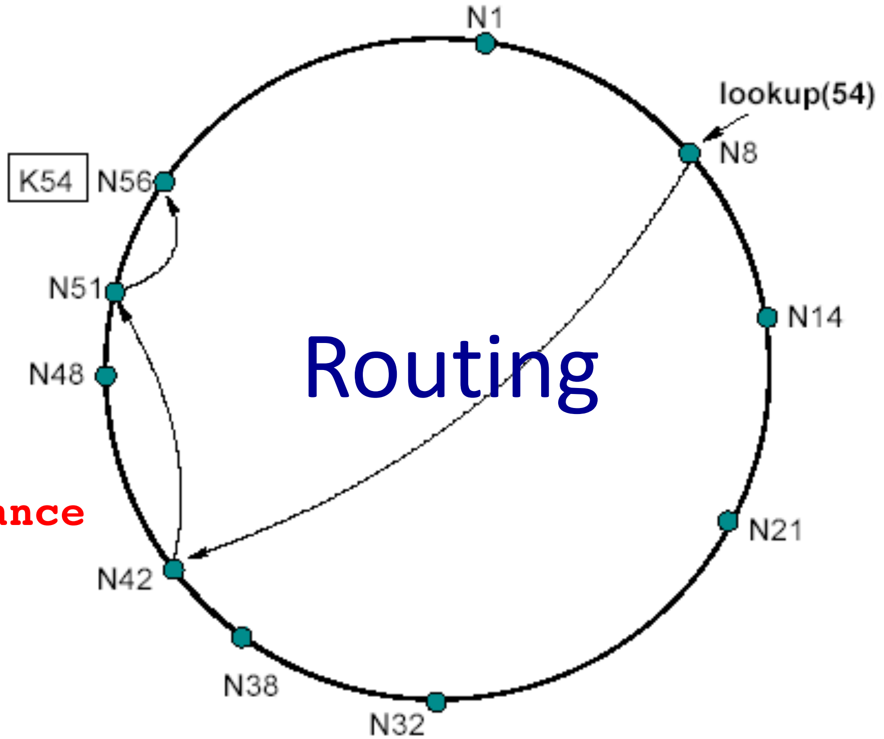
```
lookup (id):
  if ( id > pred.id &&
       id <= my.id )
     return my.id;
  else
     return succ.lookup(id);
```



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

Efficient lookup in Chord

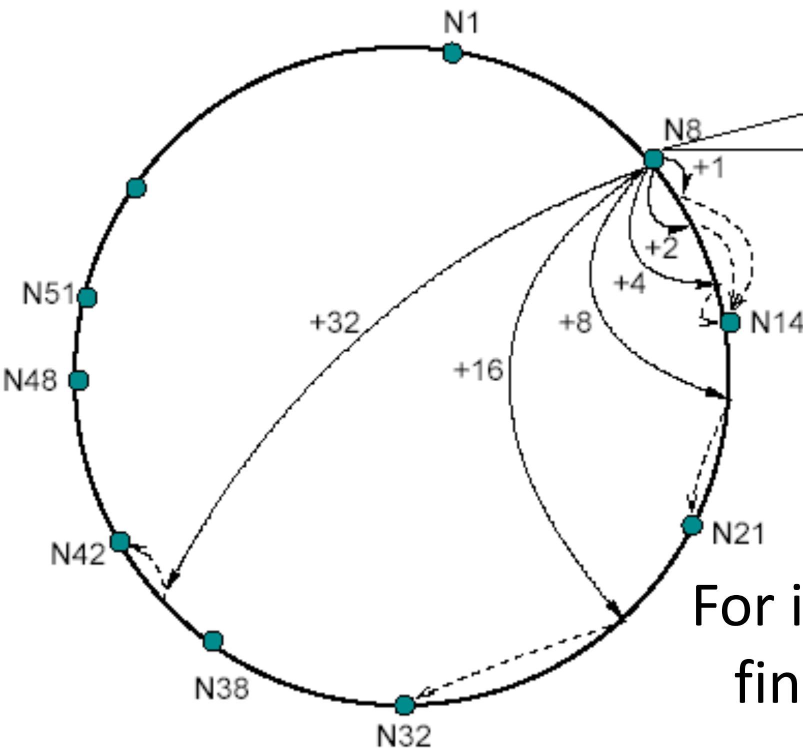
```
lookup (id):
  if ( id > pred.id &&
       id <= my.id )
return my.id;
else
     // fingers() by decreasing distance
for finger in fingers():
    if id >= finger.id
    return finger.lookup(id);
return succ.lookup(id);
```



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

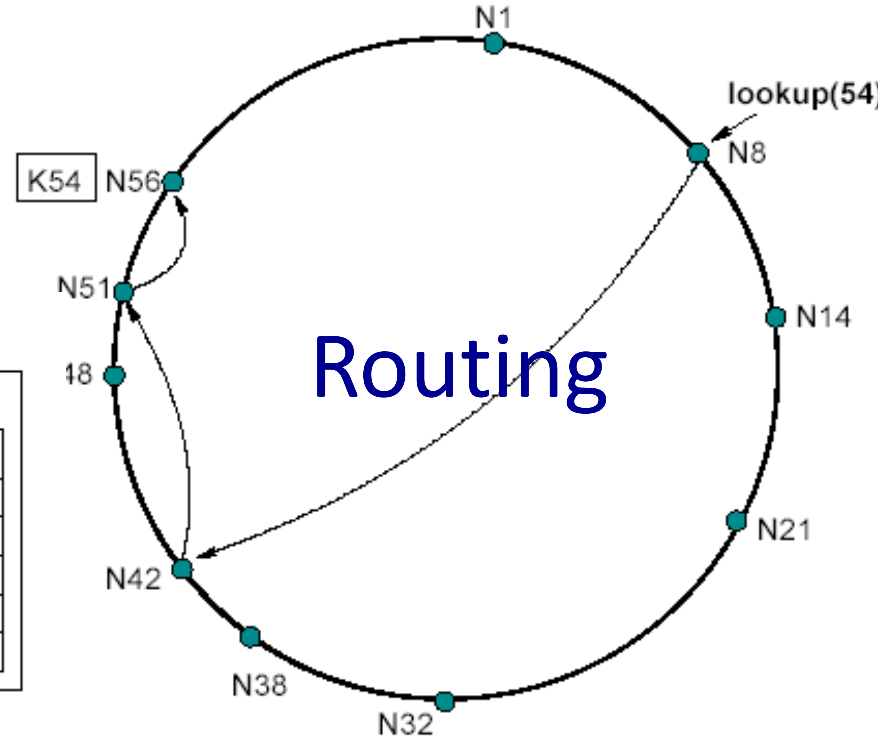
Building routing tables

Routing Tables



Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

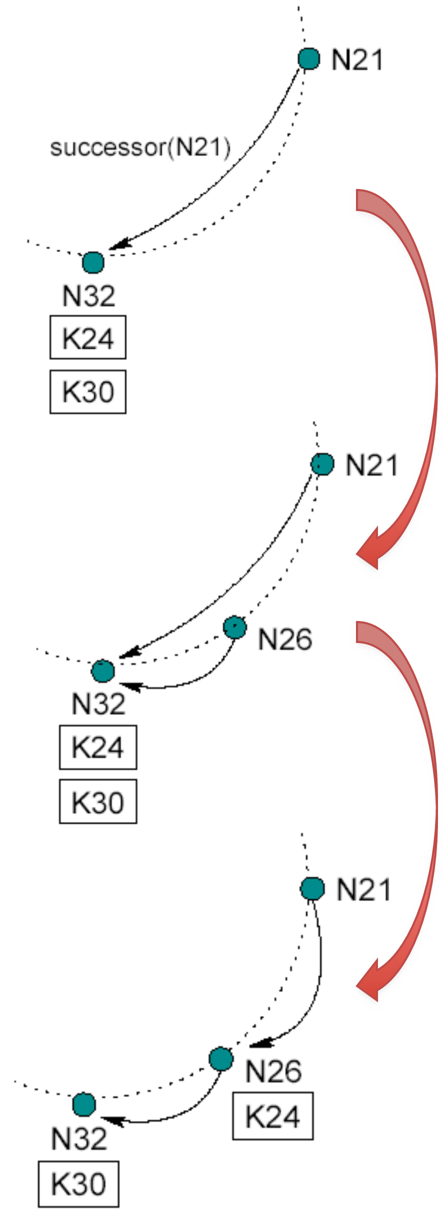


For i in 0...log n:

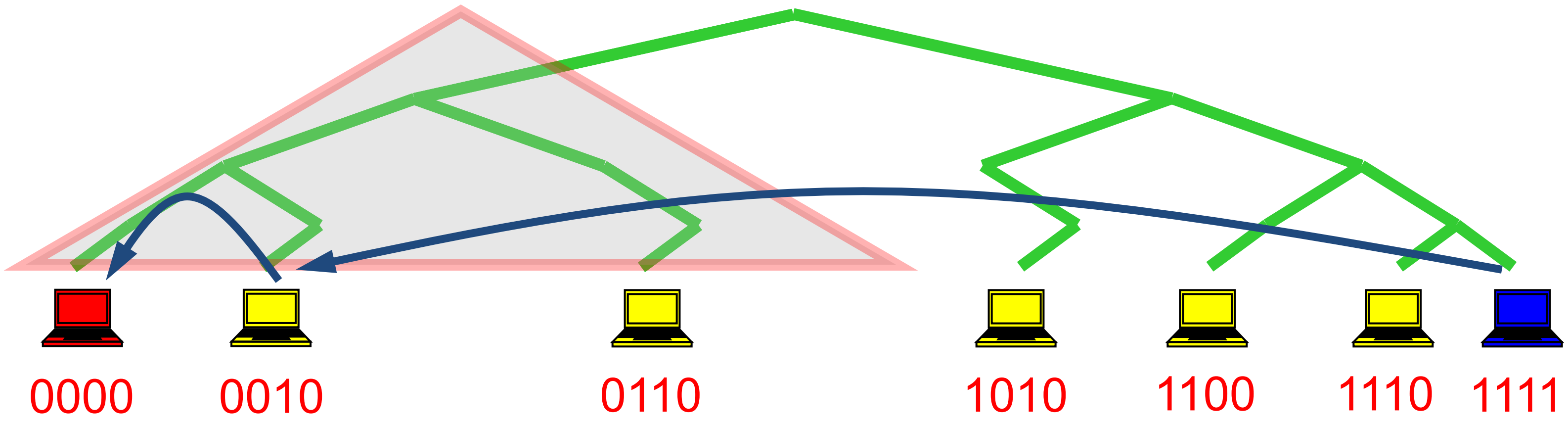
finger[i] = *successor* ($(\text{my.id} + 2^i) \bmod 2^{160}$)

Joining and managing routing

- Join:
  - Choose nodeid
  - *Lookup (my.id)* to find place on ring
  - During lookup, discover future successor
  - Learn predecessor from successor
  - Update succ and pred that you joined
  - Find fingers by *lookup* (($my.id + 2^i$) mod 2^{160})
- Monitor:
  - If doesn't respond for some time, find new
- Leave: Just go, already!
  - (Warn your neighbors if you feel like it)



Performance optimizations



- Routing entries need not be drawn from strict distribution as finger algorithm shown
  - Choose node with lowest latency to you
  - Will still get you ~ ½ closer to destination
- Less flexibility in choice as closer to destination

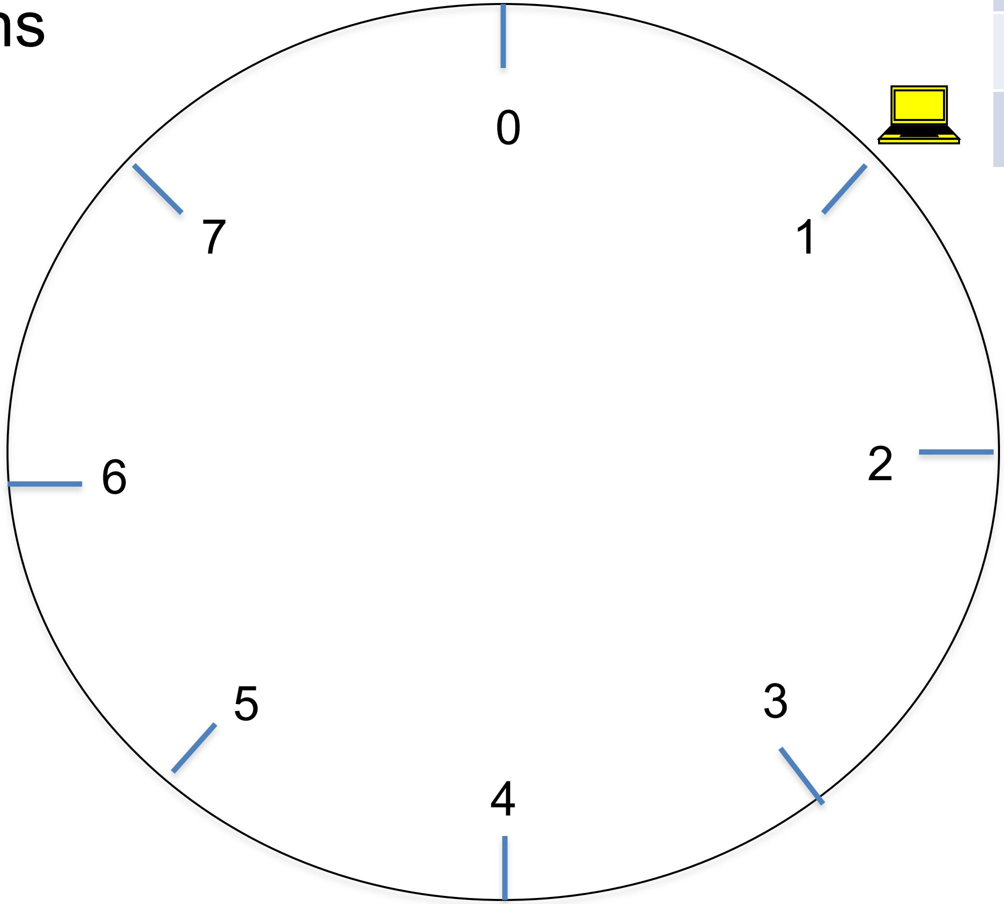
DHT Design Goals

- An “overlay” network with:
  - Flexible mapping of keys to physical nodes
  - Small network diameter
  - Small degree (fanout)
  - Local routing decisions
  - Robustness to churn
  - Routing flexibility
  - Decent locality (low “stretch”)
- Different “storage” mechanisms considered:
  - Persistence w/ additional mechanisms for fault recovery
  - Best effort caching and maintenance via soft state

Chord DHT Example

- Assume an identifier space [0..8]
- Node n1 joins

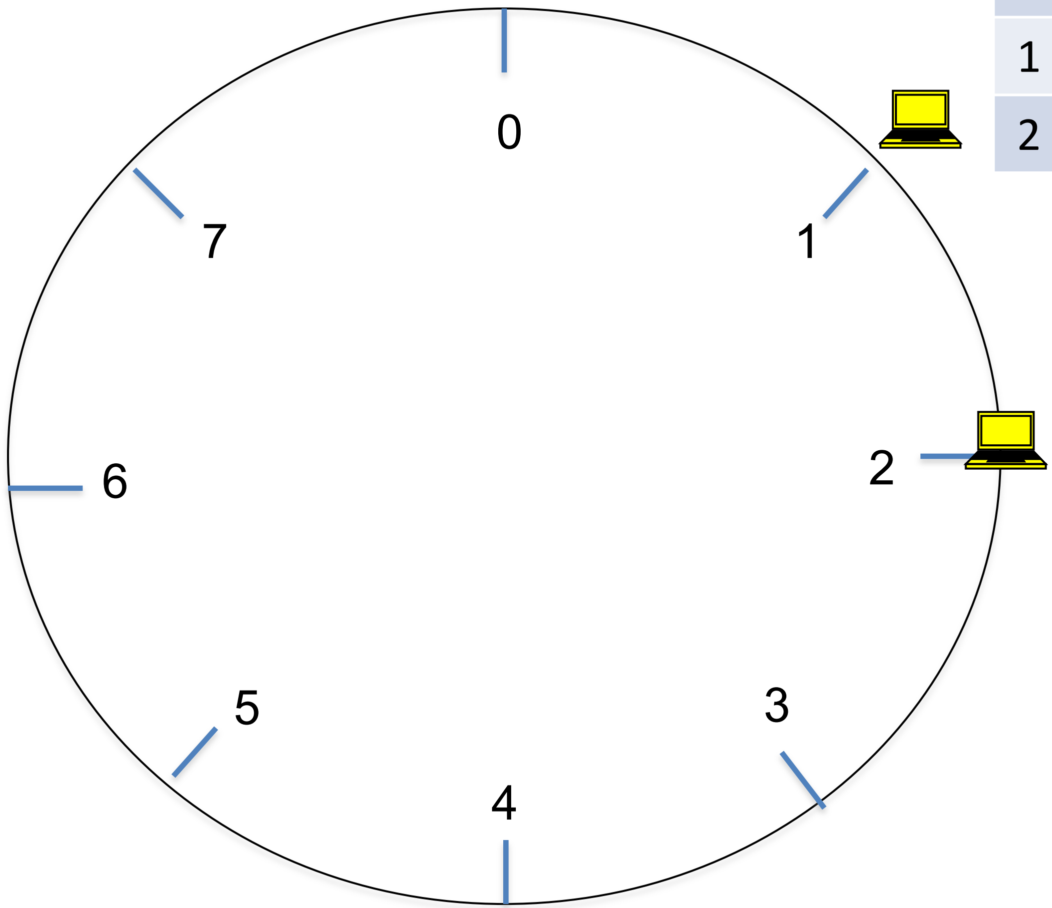
i	id+2^i	succ
0	2	1
1	3	1
2	5	1



Chord DHT Example

- Node n2 joins

i	id+2^i	succ
0	2	2
1	3	1
2	5	1



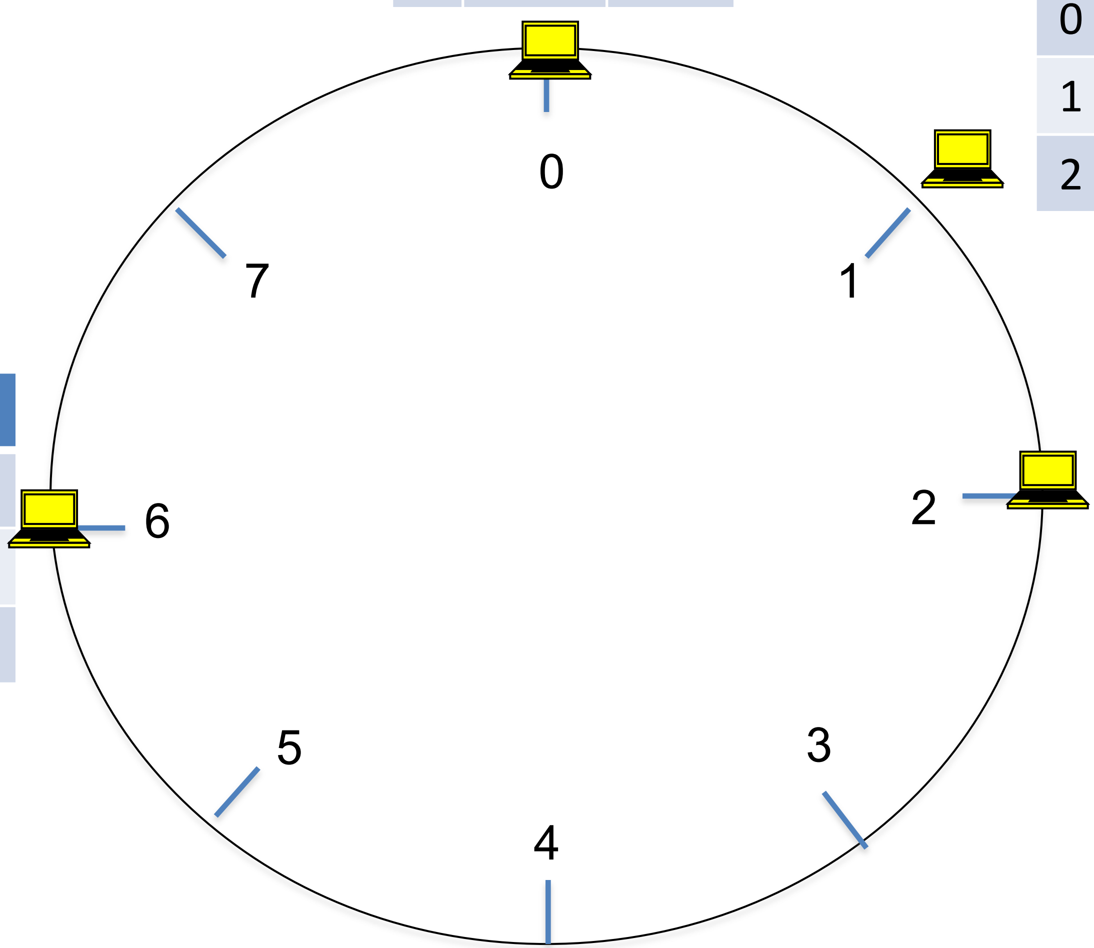
i	id+2^i	succ
0	3	1
1	4	1
2	6	1

Chord [illegible]ample

- Nodes n0 and n6 join

i	id+2^i	succ
0	1	1
1	2	2
2	4	6

i	id+2^i	succ
0	2	2
1	3	6
2	5	6



i	id+2^i	succ
0	7	0
1	0	0
2	2	2

i	id+2^i	succ
0	3	6
1	4	6
2	6	6

Chord ... le

- Nodes: n1, n2, n0, n6
- Items: f7, f1

i	id+2^i	succ
0	1	1
1	2	2
2	4	6

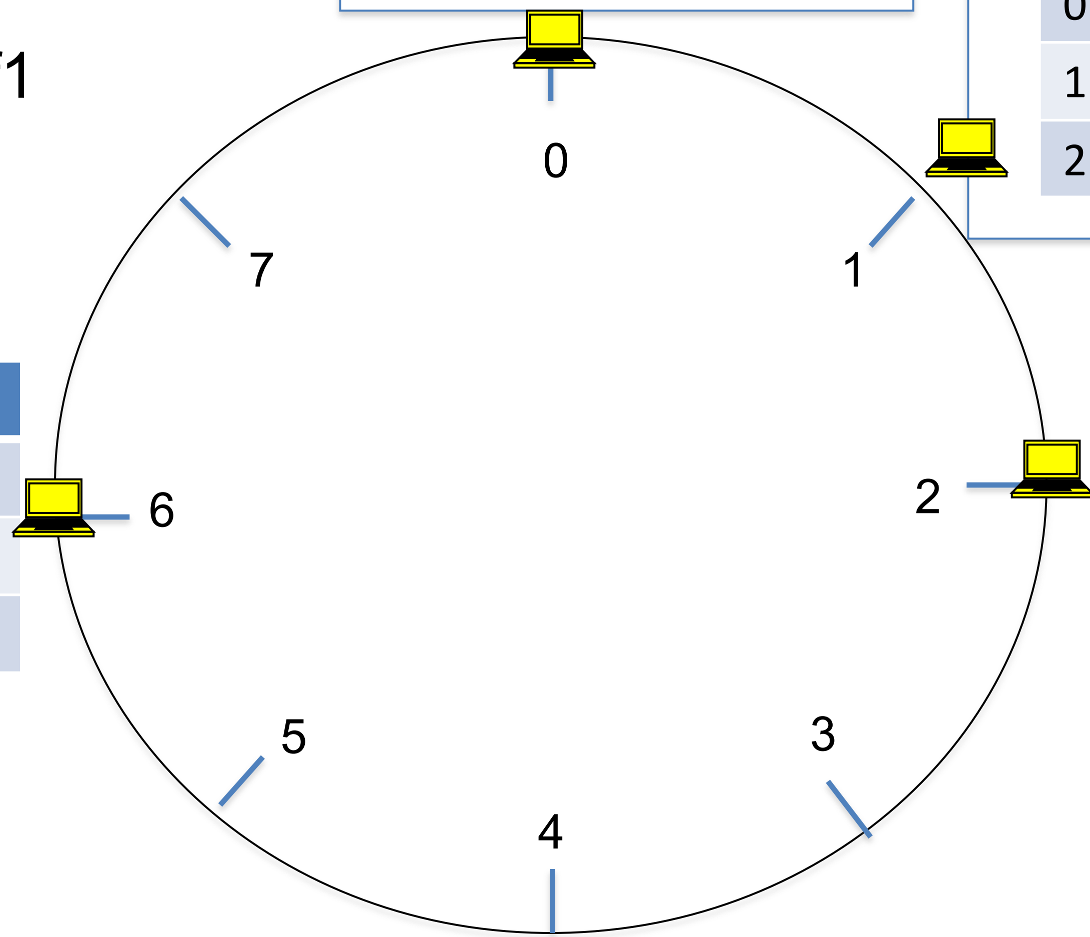
Items

7

i	id+2^i	succ
0	2	2
1	3	6
2	5	6

Items

1



i	id+2^i	succ
0	7	0
1	0	0
2	2	2

i	id+2^i	succ
0	3	6
1	4	6
2	6	6

Chord le

- Upon receiving a query for item id
- A node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id

i	id+2^i	succ
0	1	1
1	2	2
2	4	6

Items

7

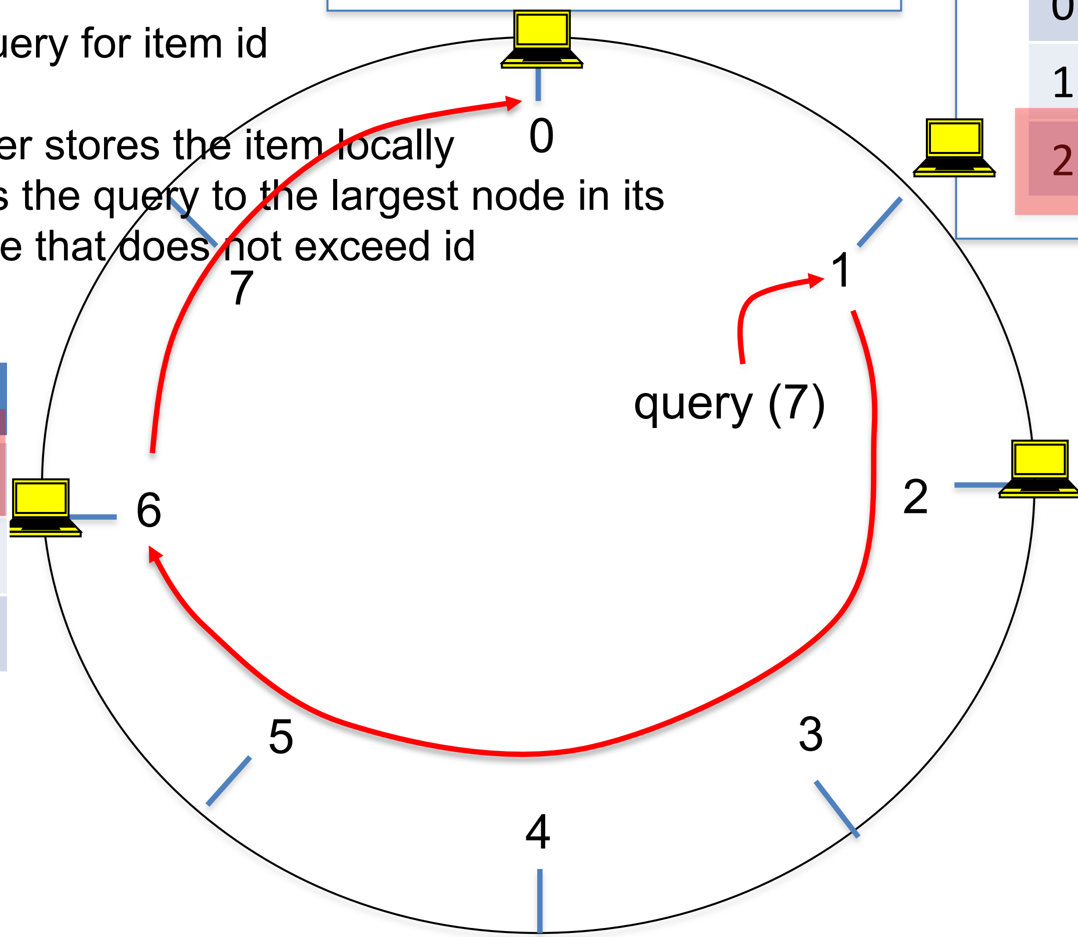
i	id+2^i	succ
0	2	2
1	3	6
2	5	6

Items

1

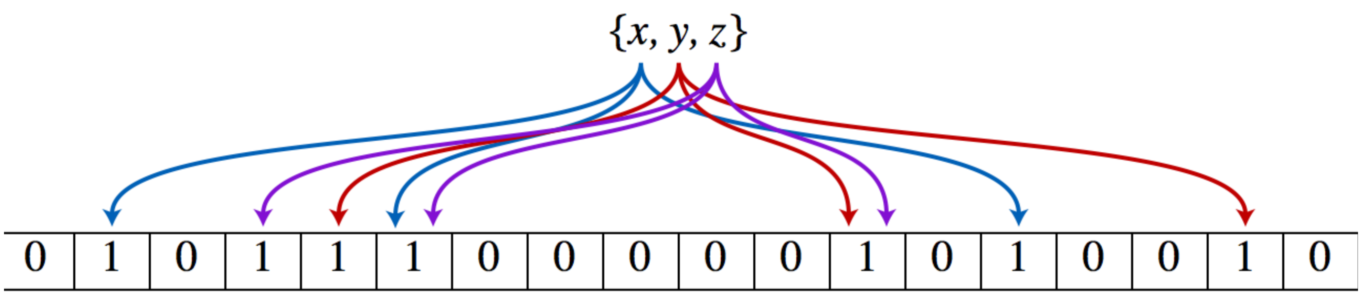
i	id+2^i	succ
0	3	6
1	4	6
2	6	6

i	id+2^i	succ
0	7	0
1	0	0
2	2	2



Bloom Filters

- Data structure for probabilistic membership testing
  - Small amount of space, constant time operations
  - False positives possible, no false negatives
  - Useful in per-flow network statistics, sharing information between cooperative caches, etc.
- Basic idea using hash fn's and bit array
  - Use k independent hash functions to map item to array
  - If all array elements are 1, it's present. Otherwise, not



Bloom Filters

Start with an m bit array, filled with 0s.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

To insert, hash each item k times. If $H_i(x) = a$, set $Array[a] = 1$.

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

To check if y is in set, check array at $H_i(y)$. All k values must be 1.

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Possible to have a false positive: all k values are 1, but y is not in set.

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Summary

- Peer-to-peer systems
  - Unstructured systems
    - Finding hay, performing keyword search
  - Structured systems (DHTs)
    - Finding needles, exact match
- Distributed hash tables
  - Based around consistent hashing with views of O(log n)
  - Chord, Pastry, CAN, Koorde, Kademlia, Tapestry, Viceroy, …
- Lots of systems issues
  - Heterogeneity, storage models, locality, churn management, underlay issues, …
  - DHTs deployed in wild: Vuze (Kademlia) has 1M+ active users