A distributed system is:

- a collection of *independent computers* that appears to its users as a *single coherent system*

Components need to:

- Communicate
  - one-to-one (Request/Reply, RPC)
  - one-to-many communication (epidemic)
- Cooperate => support needed
  - **Naming** – enables some resource sharing
  - Synchronization

Problem solved by a name service

Given a [unstructured] name (i.e., the ‘key’)

… locate/return the associated value

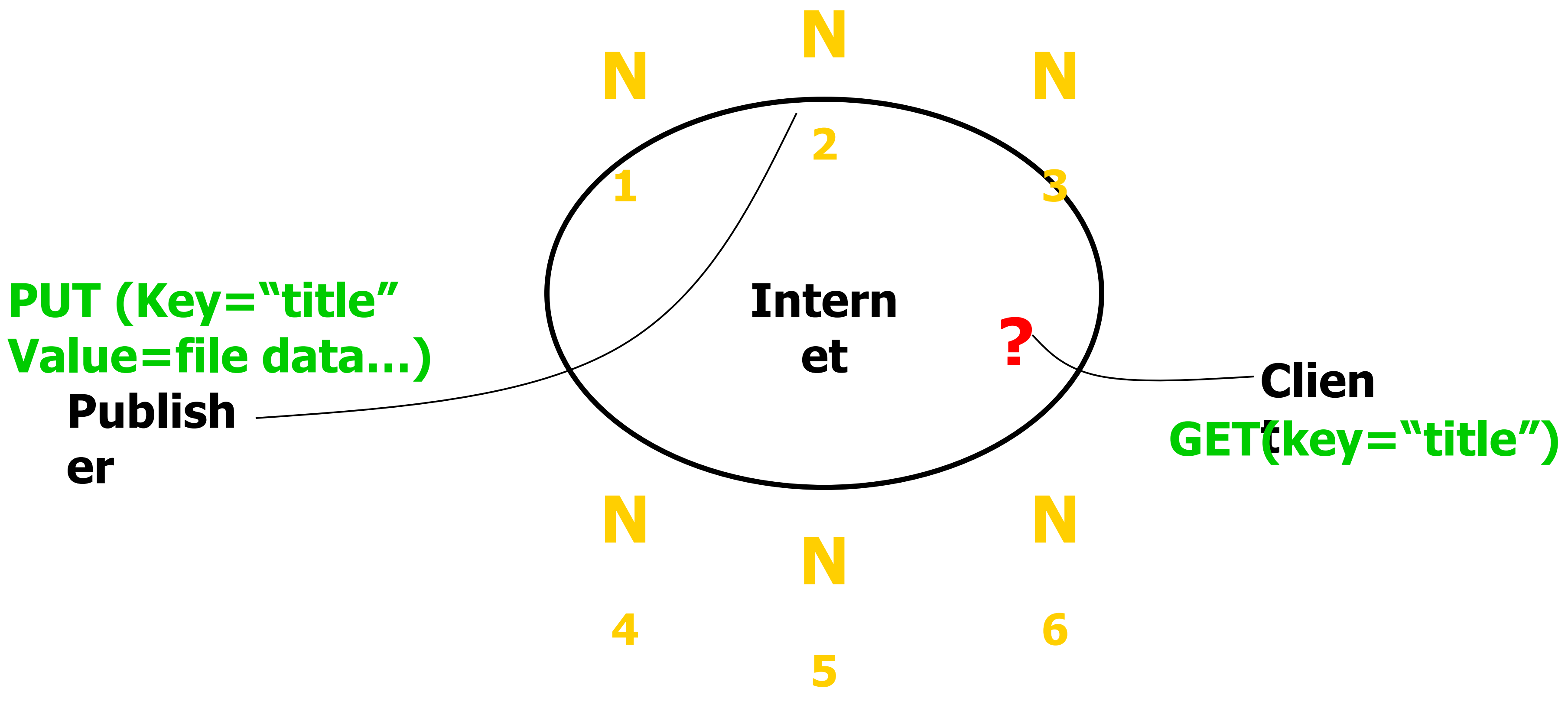
(or the node responsible for the value)

Similar to a [huge] hash-table:

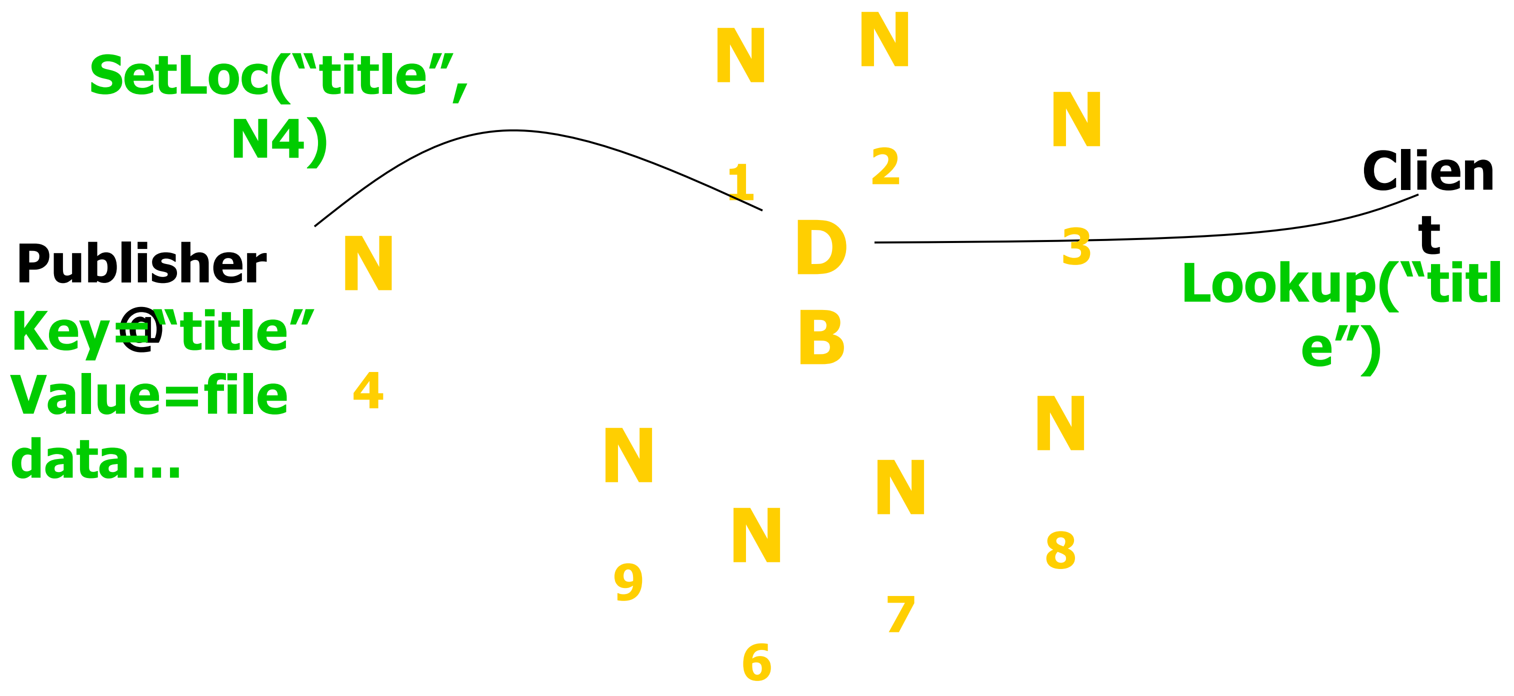
- API: put (key, value),
  get (key) ☐ value

The Lookup Problem

At the heart of many services



Strawman1: Centralized Lookup (Napster)

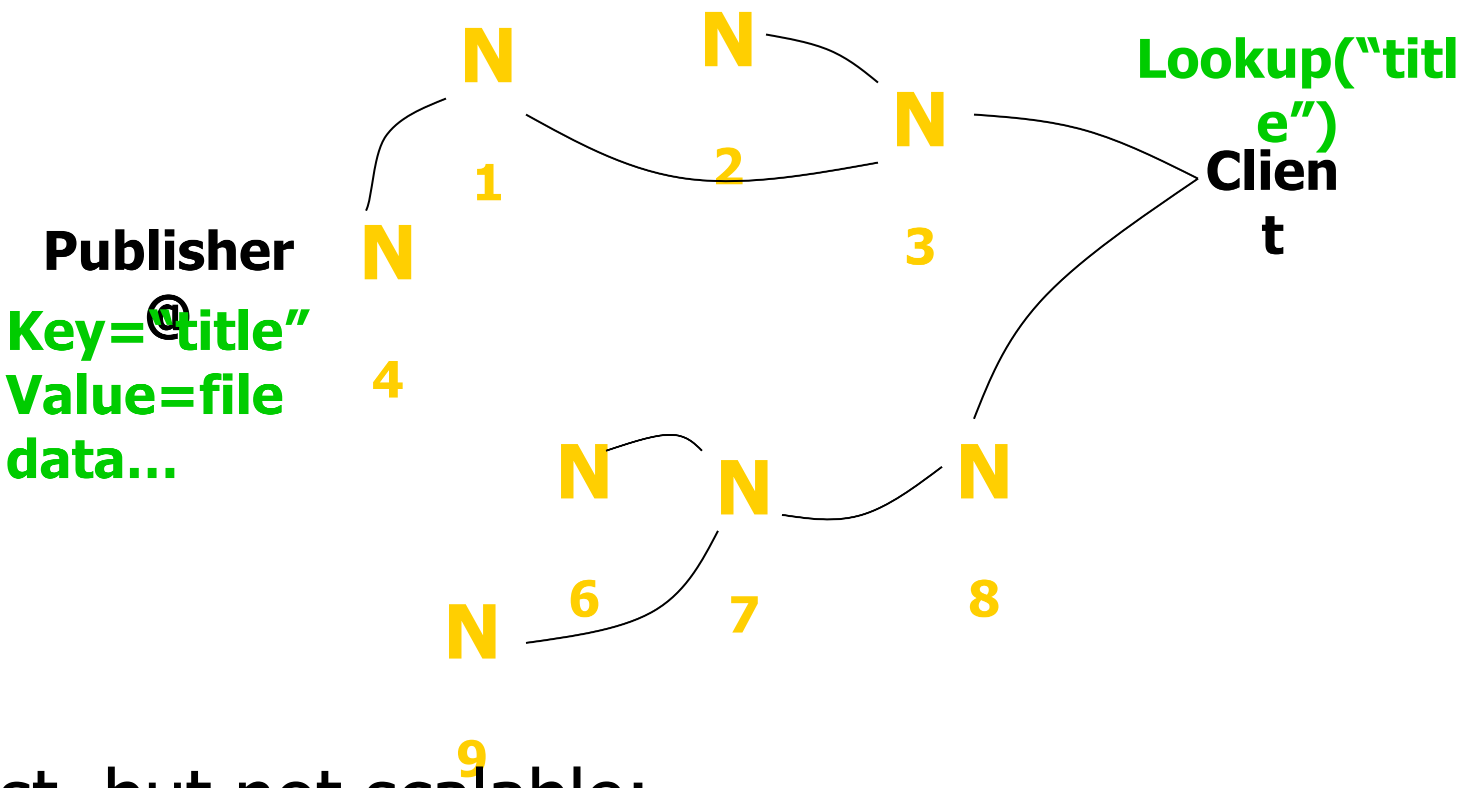


Simple, but:

O(N) state, and

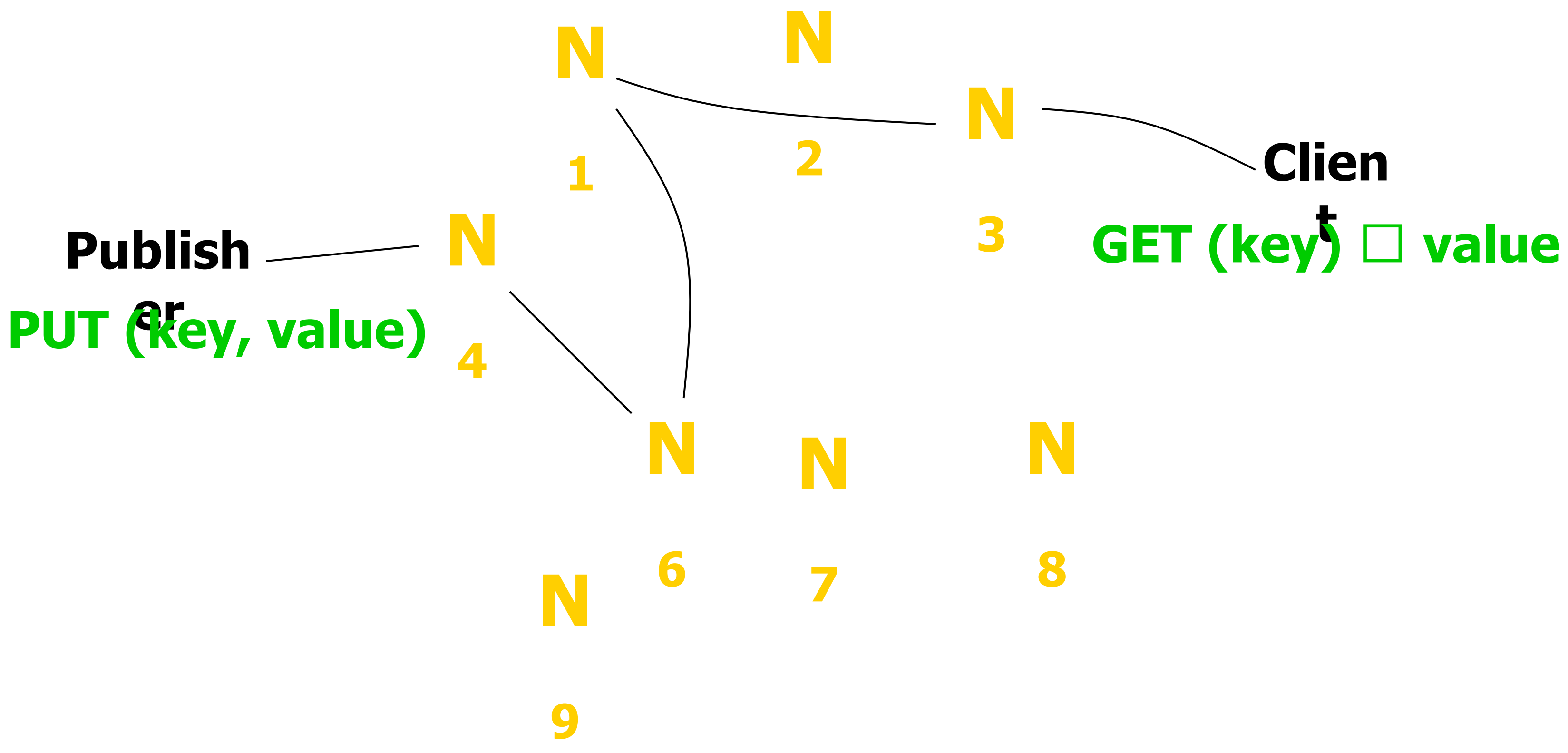
a single point of failure

Strawman2: Flooded Queries (Gnutella)



Robust, but not scalable:
worst case O(N) messages per lookup

A way to think about the solution:
decide on a rendez-vous point between publisher (PUT) and client (GET) based on key



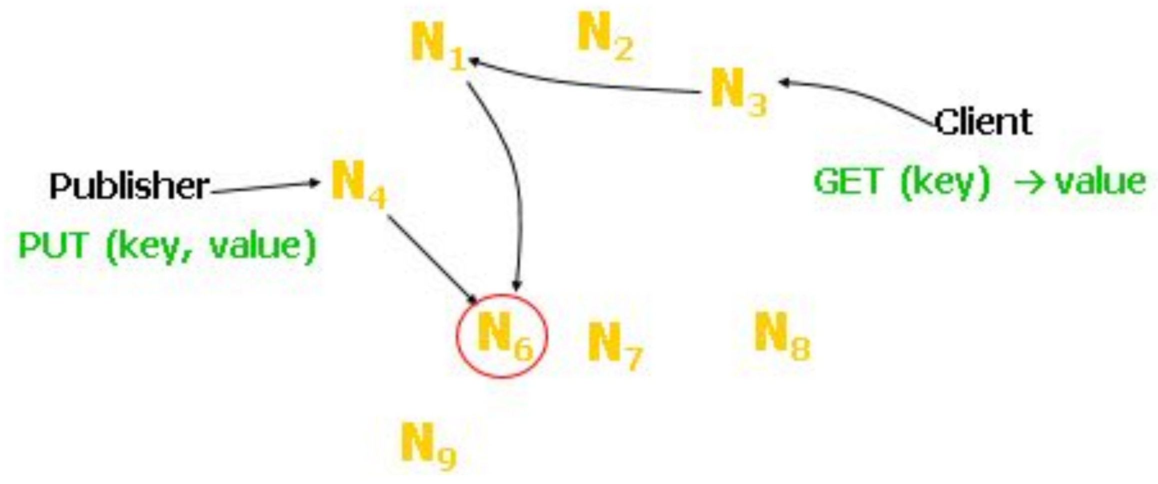
Note: problem is simpler if routing info directly encoded in key

Desirable Properties

- Scalable: multiple axes - network traffic overhead, state at nodes, routing cost, repair cost,
  - Incremental scalability
- Efficient: find items quickly (latency)
- Dynamic: deals with node failure, join
- General-purpose: unstructured keys
- Decentralized / symmetric design
  - i.e., nodes have similar roles

Design Issues

- **Issue 1: How to map keys to nodes?**
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity



Mapping keys to nodes (Strawman Solutions)

Assumption: All nodes know <u>ALL</u> other nodes <u>(Local State: O(N))</u>

Membership

N_0	N_1	N_2	N_3

Simplification: Keys are English Words

Solution 1:
Key ID = First Character
Unifo
Map

A-F	G-L	M-R	S-Z

Desired features

Balanced: No bucket has disproportionate number of objects

Smoothness: Addition/removal of bucket does not cause movement among existing buckets

Mapping keys to nodes (Strawman Solutions)

Assumption: All nodes know <u>ALL</u> other nodes <u>(Local State: O(N))</u>

Membership

N_0	N_1	N_2	N_3

Simplification: Keys are English Words

Solution 1:
Key ID = First Character
Uniform Key ID / Node Map

A-F	G-L	M-R	S-Z

Issues?

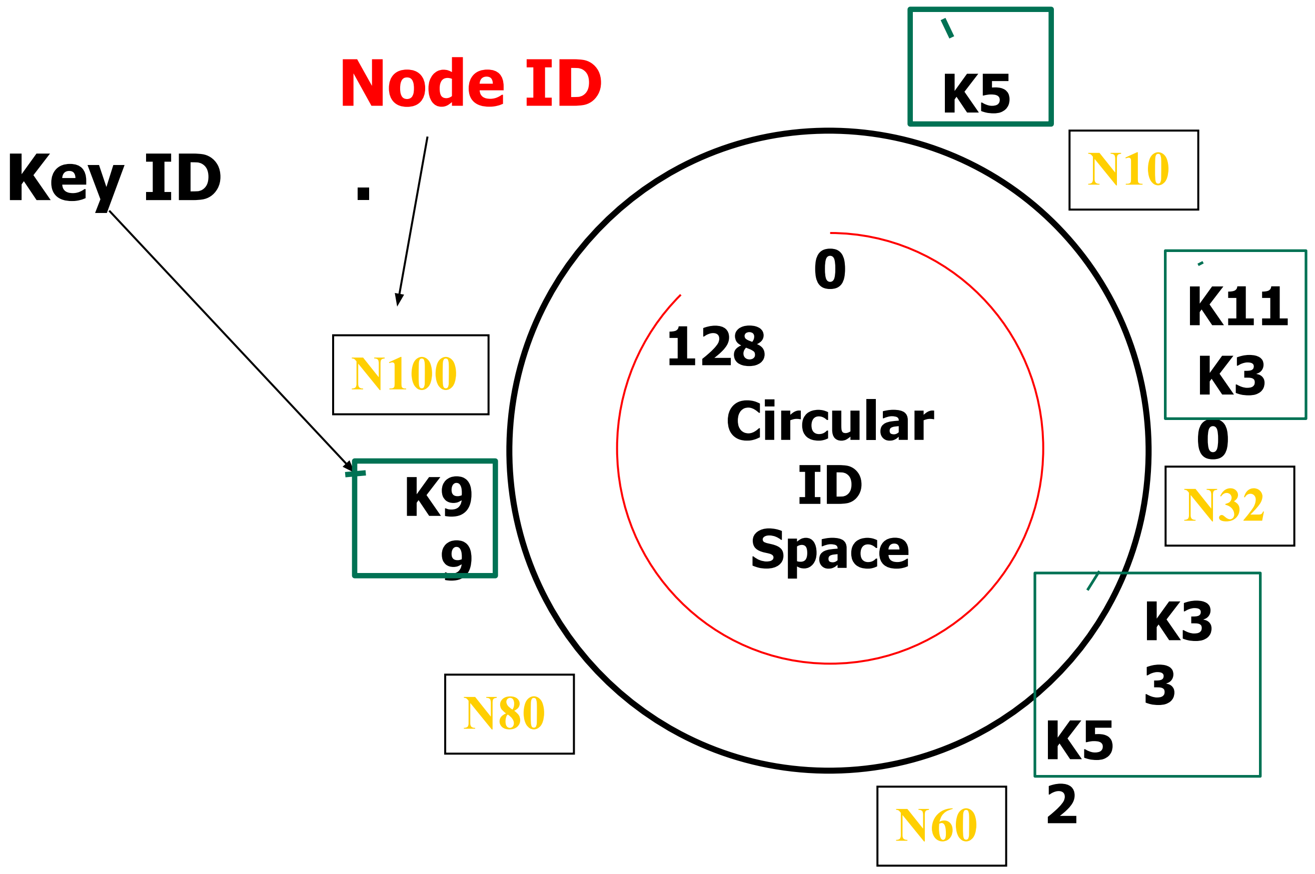
Solution 2:
Key ID = Hash(Key)
Uniform Key ID / Node Map
Node = Key ID % Nodes.Length

[0, H/4)	[H/4, H/2)	[H/2, 3*H/4)	[3*H/4, H]

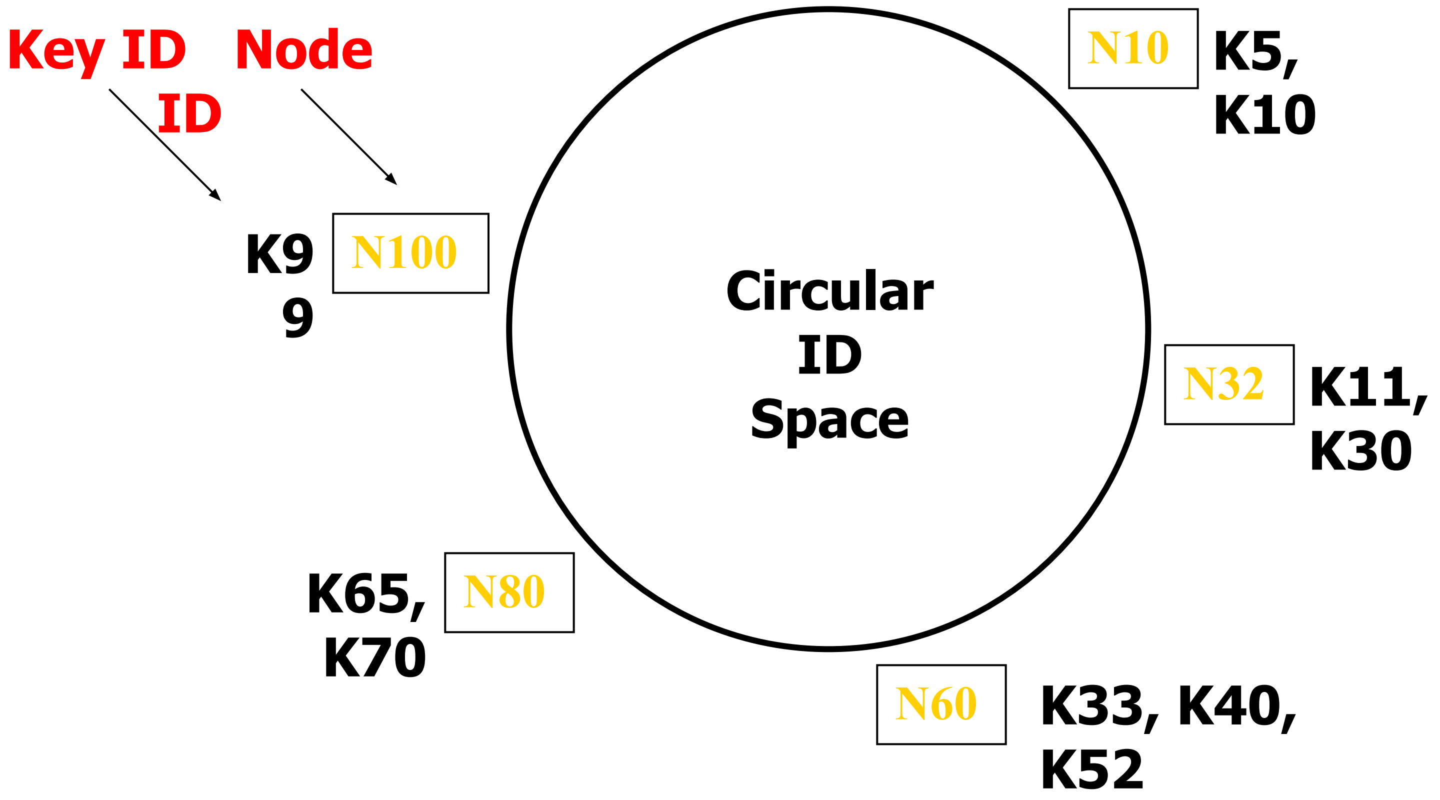
Issues?

Mapping keys to nodes

The output range of a hash function is treated as a fixed circular space or "ring".

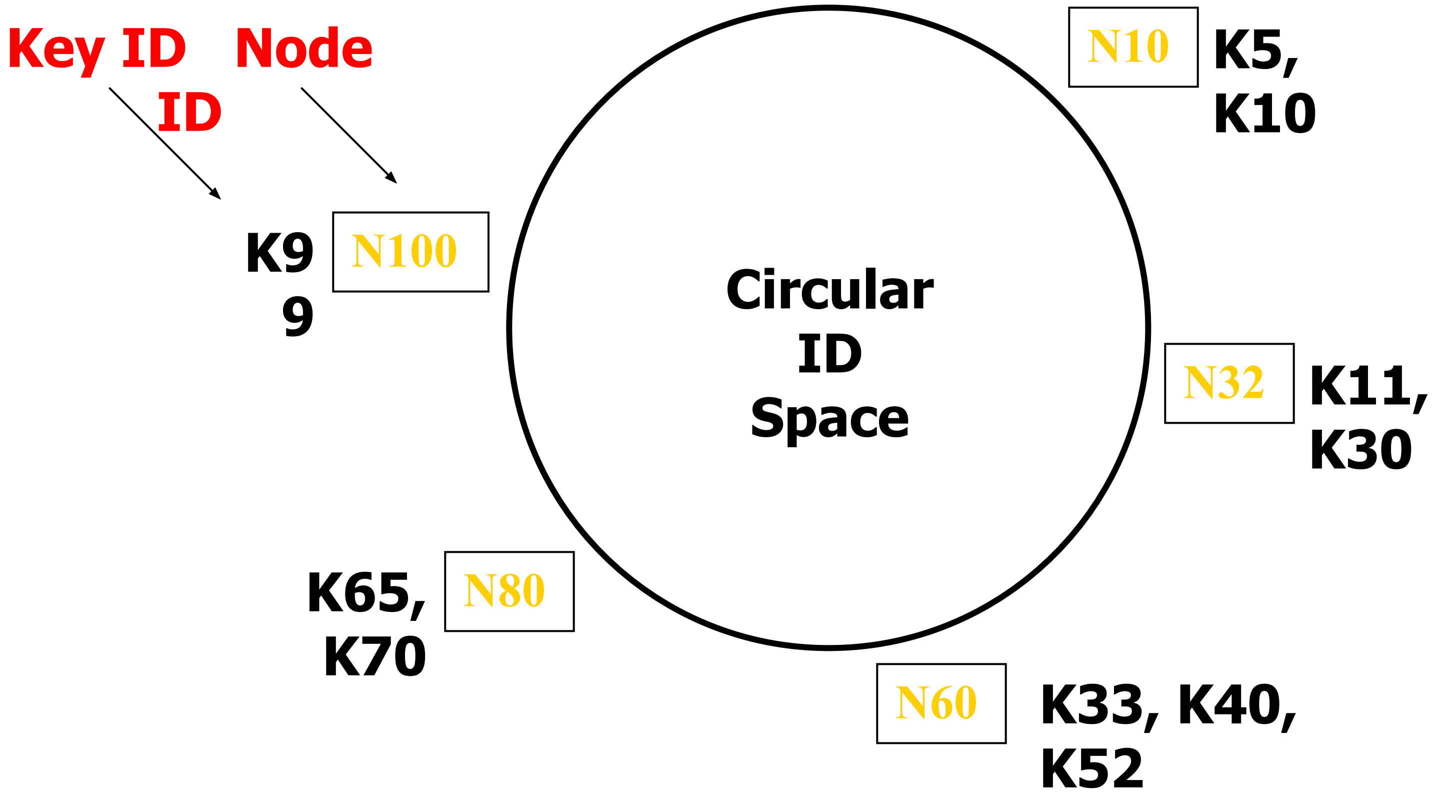


Mapping keys to nodes



Did I get?

- Load balancing
- Smoothness



- Given *K* items, and *N* machines. What is the expected load (num keys) of each machine?
  - K/N on average. Says nothing of request load per bucket
  - Symmetry argument. Equal probability.

- When a machine is added, the expected number of items that move to the newly added machine is
  - $\frac{K}{N+1}$

- with high probability no machine owns more than O($\frac{logN}{N}$) fraction → load balance

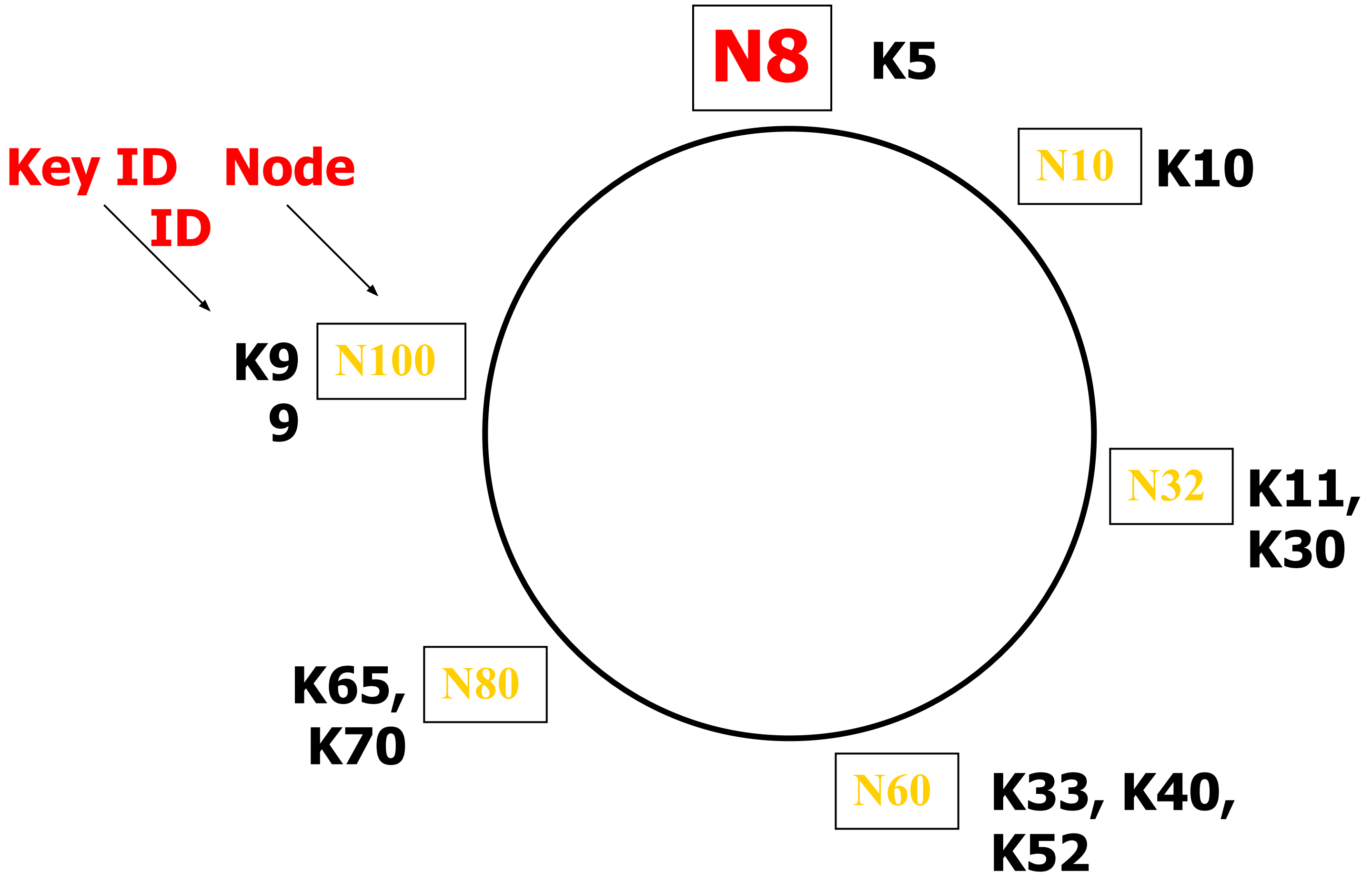
Proof

- Given *K* items, and *N* machines. What is the expected load (num keys) of each machine?
  - K/N on average. Says nothing of request load per bucket
  - Symmetry argument. Equal probability.

- When a machine is added, the expected number of items that move to the newly added machine is
  - $\frac{K}{N+1}$

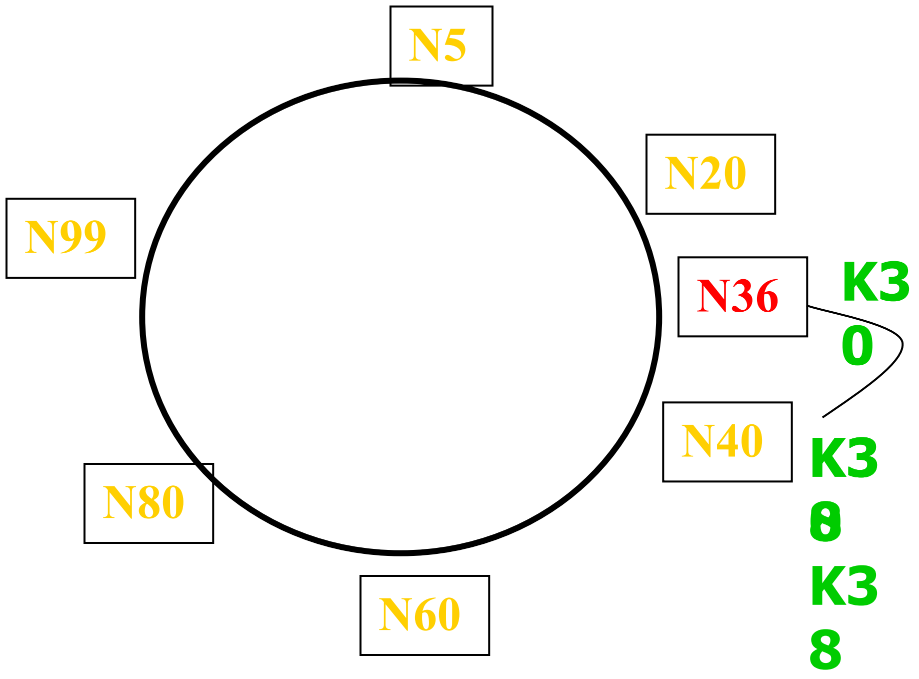
Did I get?

- Load balancing
- Smoothness



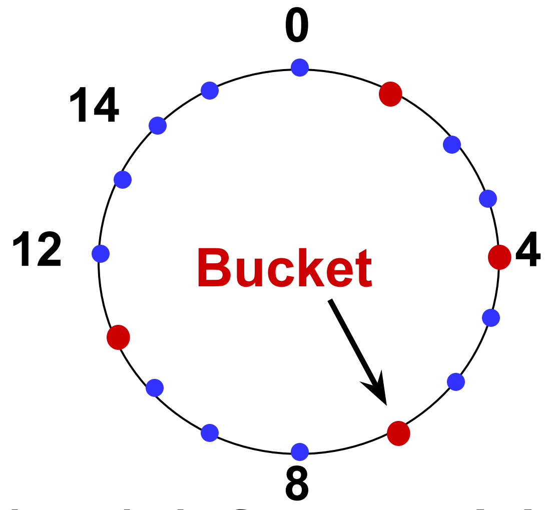
Join: Transfer Keys

- Only keys in the range are transferred



Copy keys 21..36 from N40 to N36

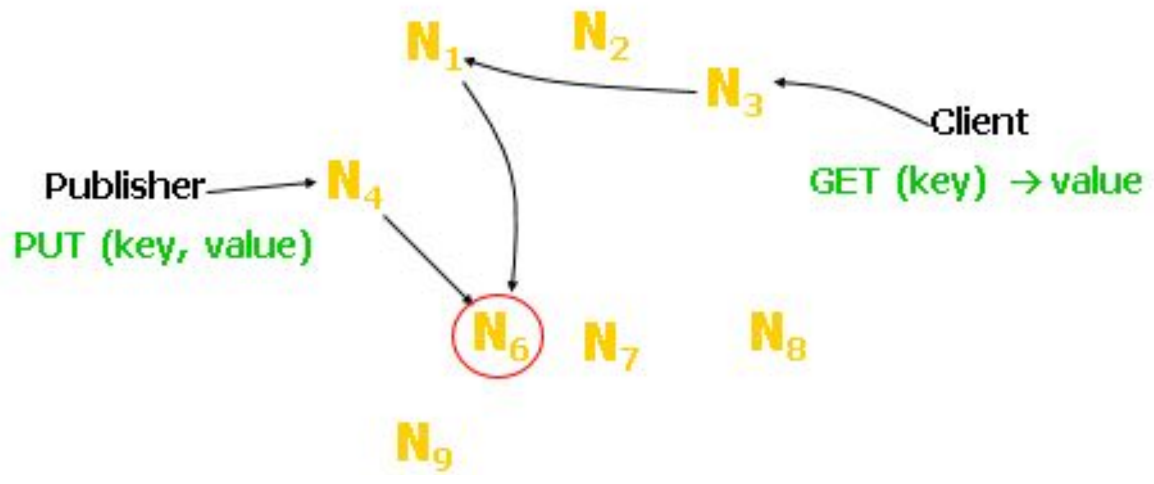
Consistent hashing and failures



- Consider network of n nodes
  - If each node has 1 bucket
    - Owns 1/nth of keyspace *in expectation*
- If a node fails:
  - Its *successor* takes over bucket
  - Achieves smoothness goal: Only *<u>localized</u>* shift, not O(n)
  - But now successor owns 2 buckets: keyspace of size *2/n*
- Instead, 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*

Design Issues

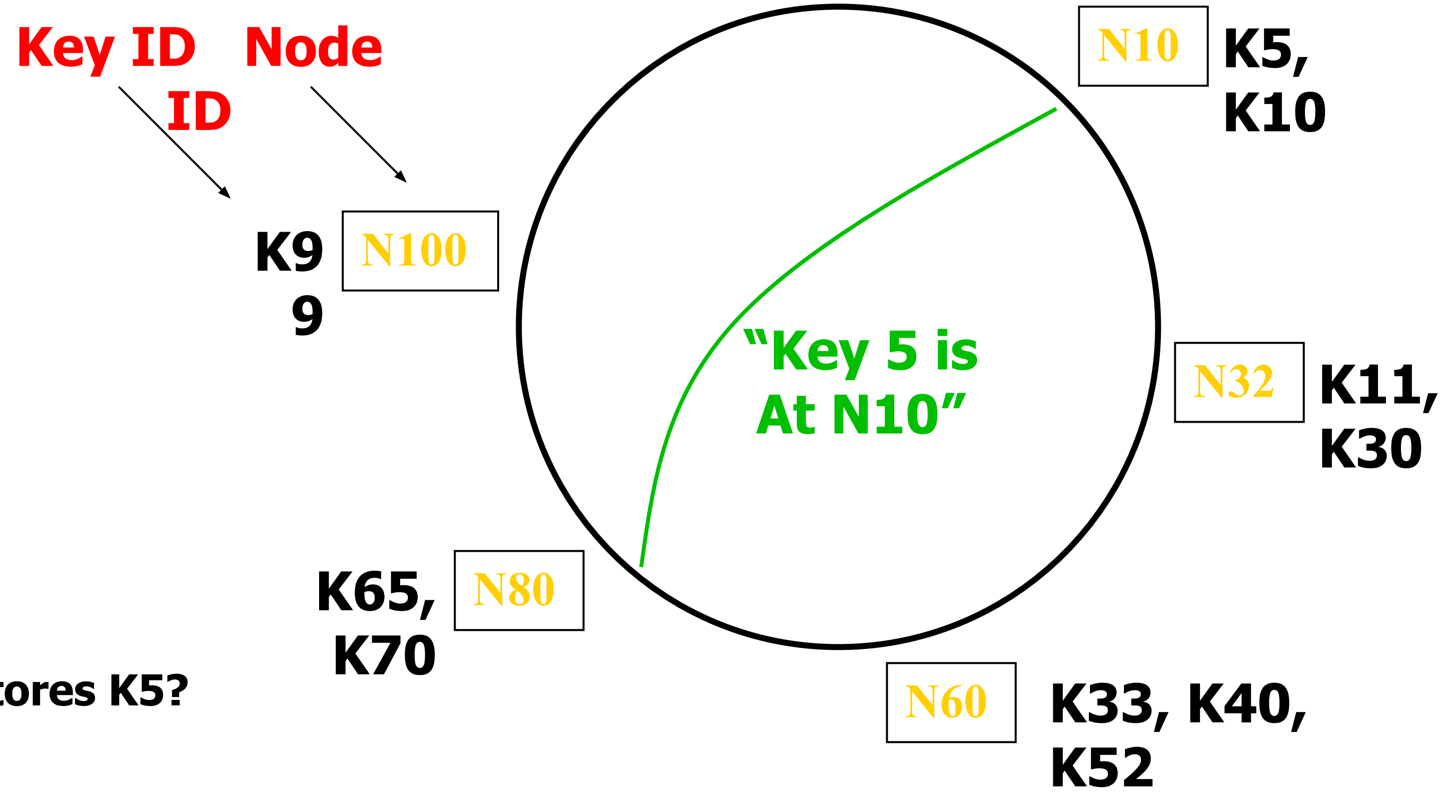
- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity



Scheme I: Consistent hashing

- Direct routing.

- Lookup cost: O(1)
- State at each node: ???

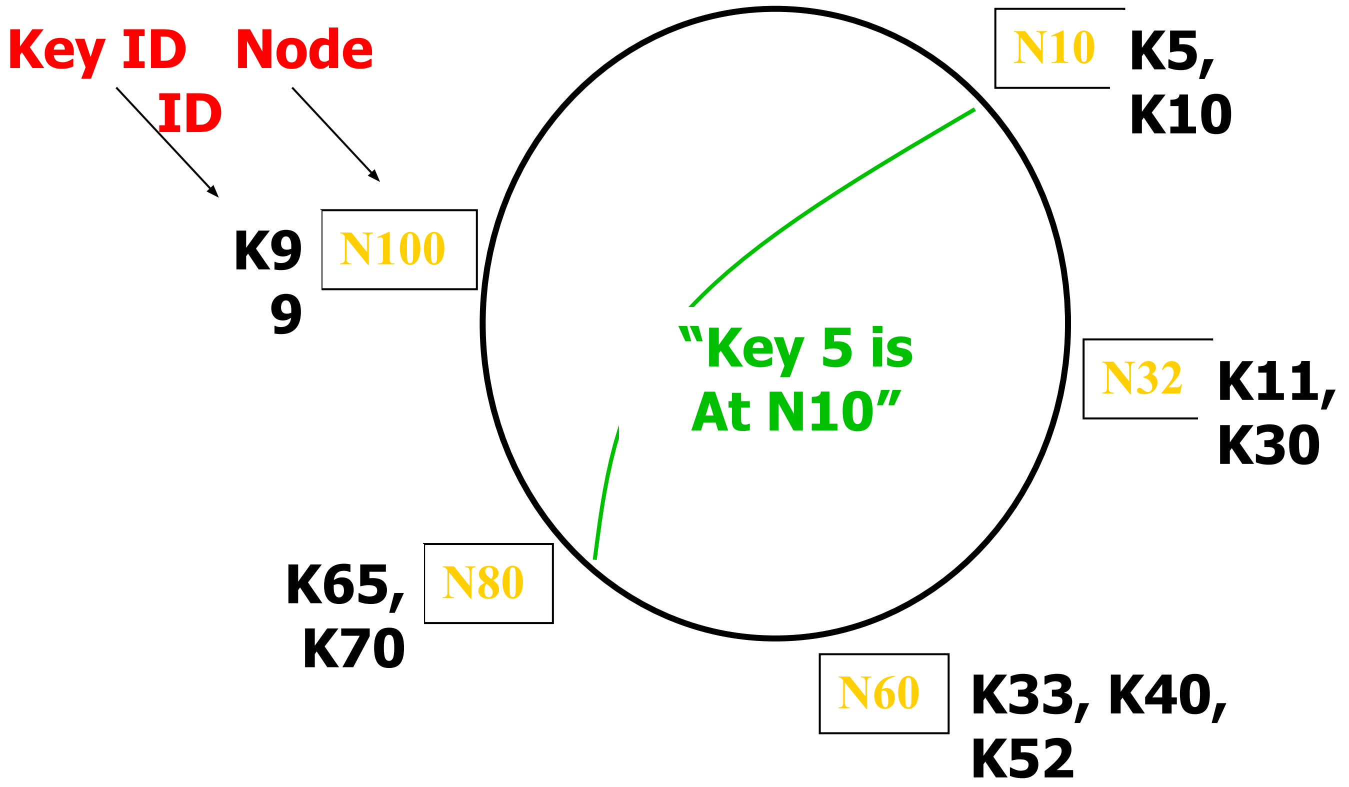


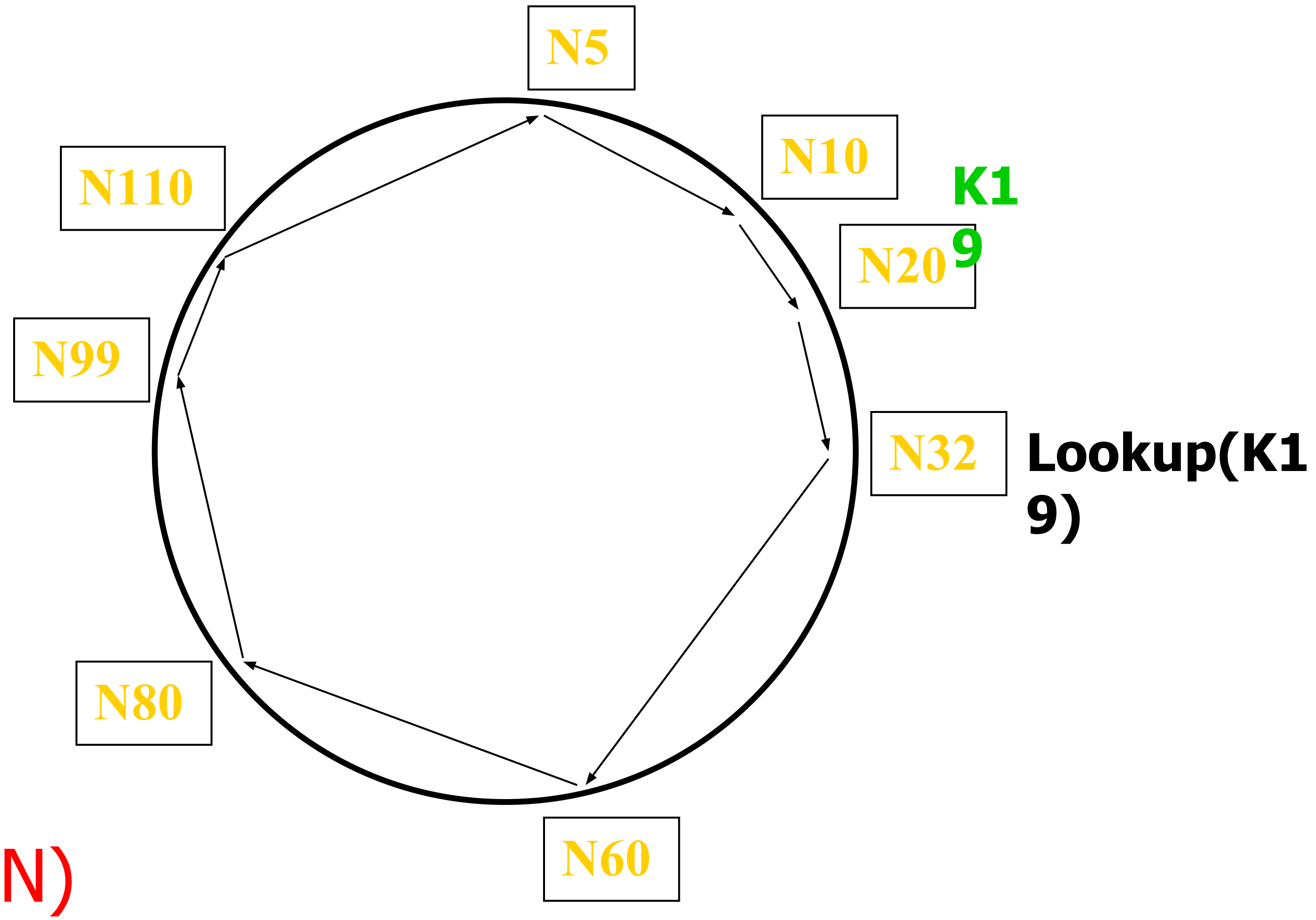
What node stores K5?

Potential Issue: Large state at each node

O(N); N number of nodes.

Can one rely on less state at each node)?





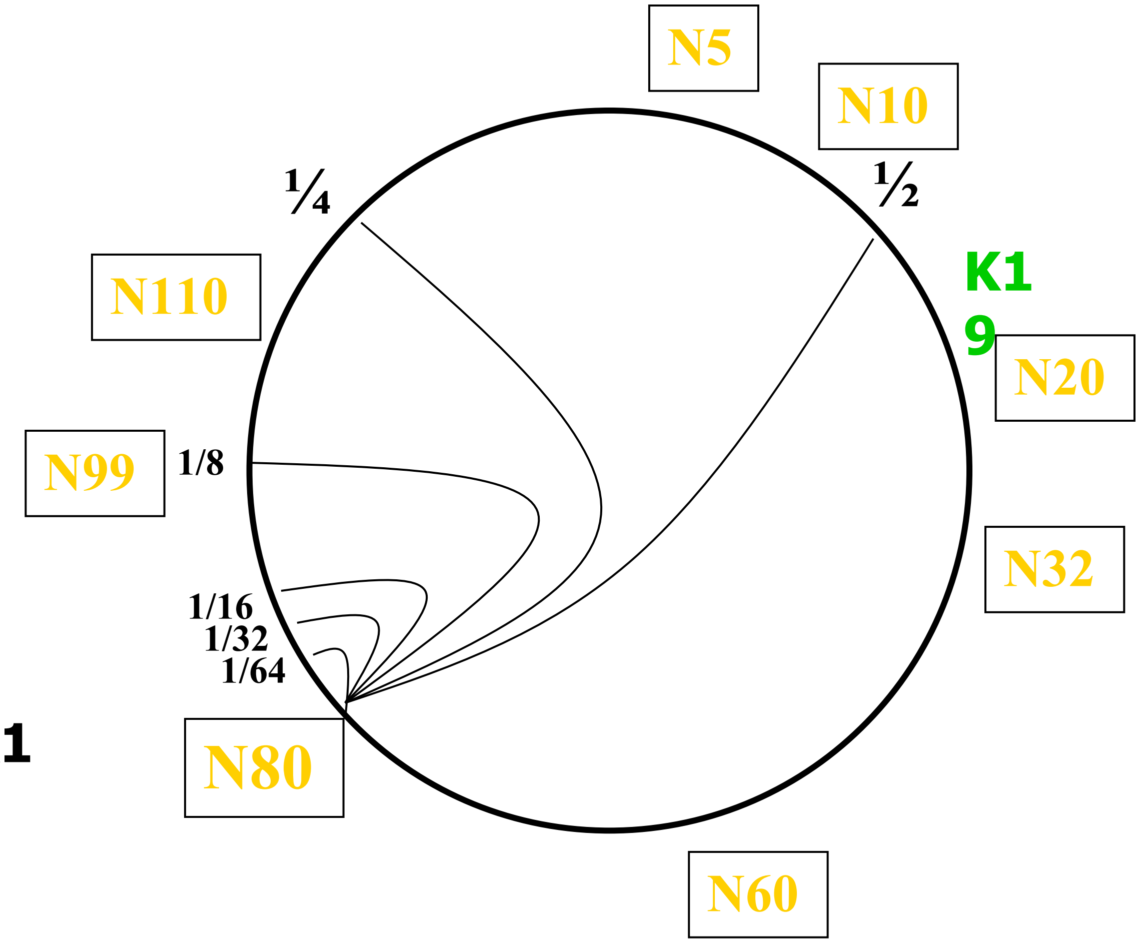
[Strawman]

Lookup: O(N)

State at each node: O(1)

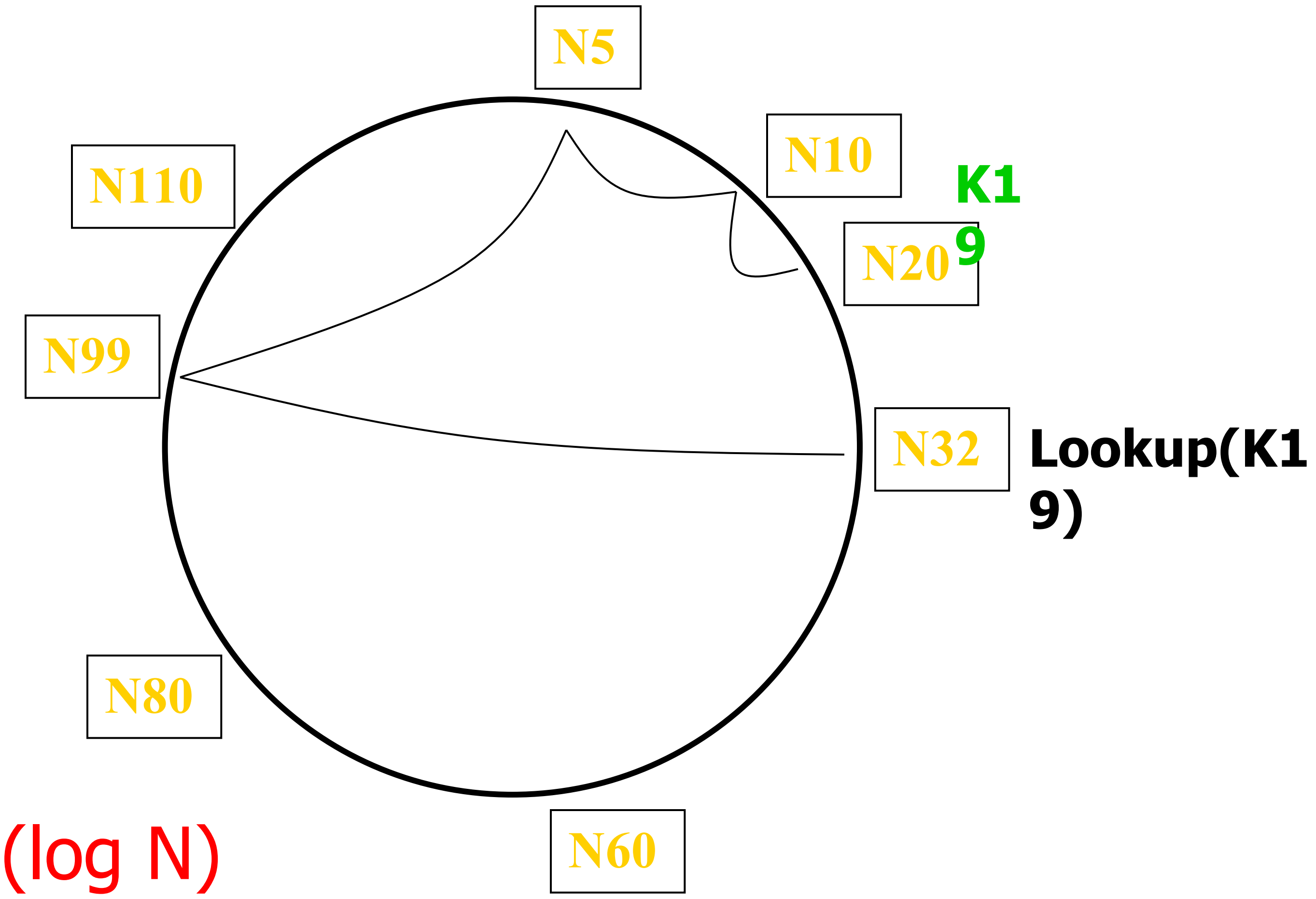
OR?

"Finger Table" Accelerates Lookups



Lookup(K19)

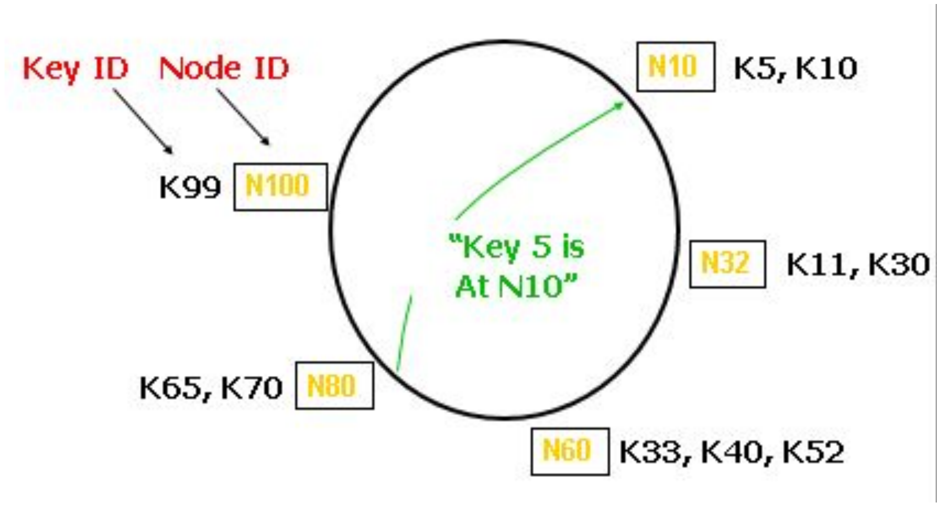
Scheme II : Distributed Hash Table

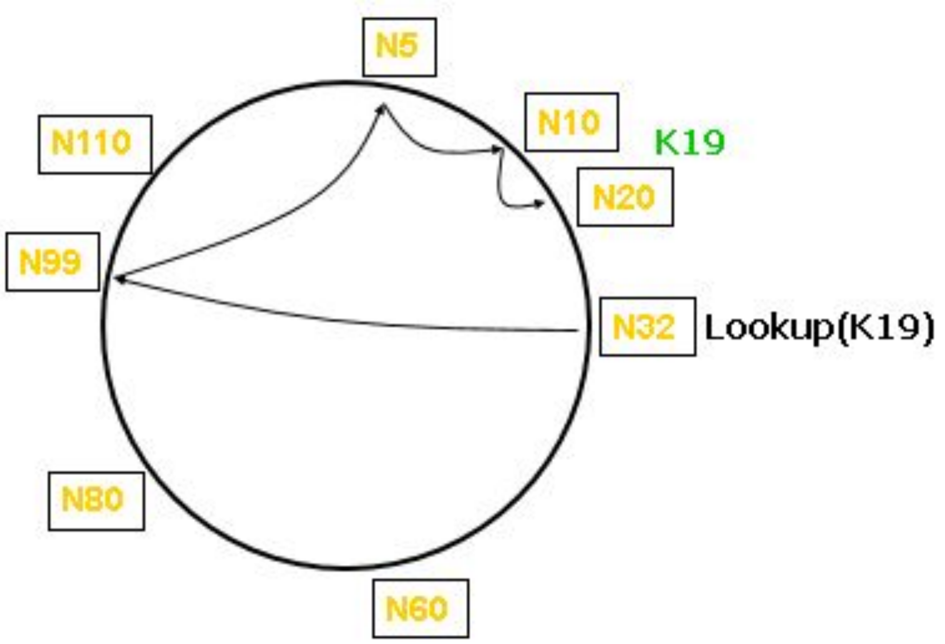


Lookup: O(log N)

State at each node: O(log N)

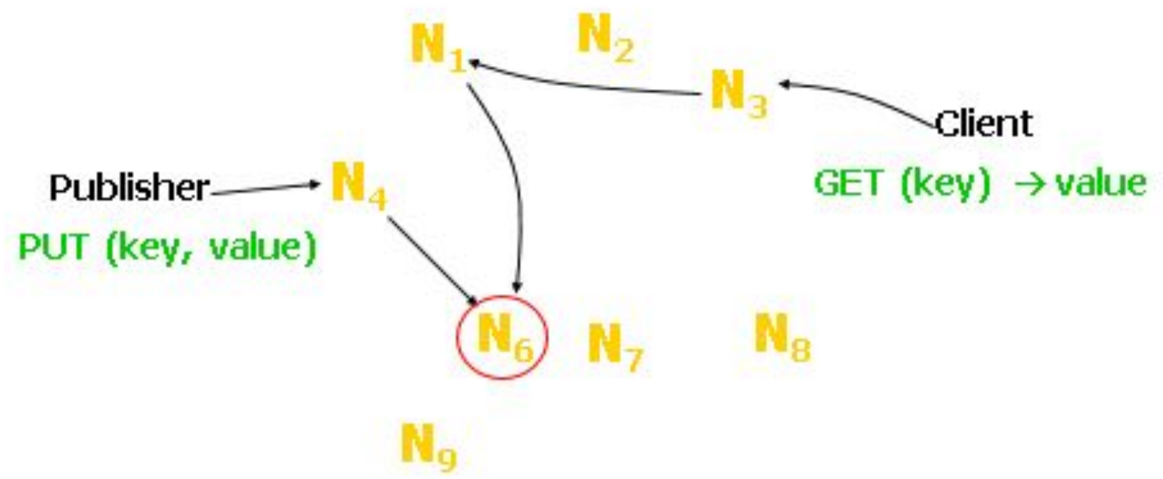
	Consistent Hashing	Distributed Hash Table
Lookup hops:	***O(1)***	*O(log N)*
Routing state per node:	***O(N)***	*O(log N)*





Design Issues

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
  - [maintain routing structure, data placement]
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity



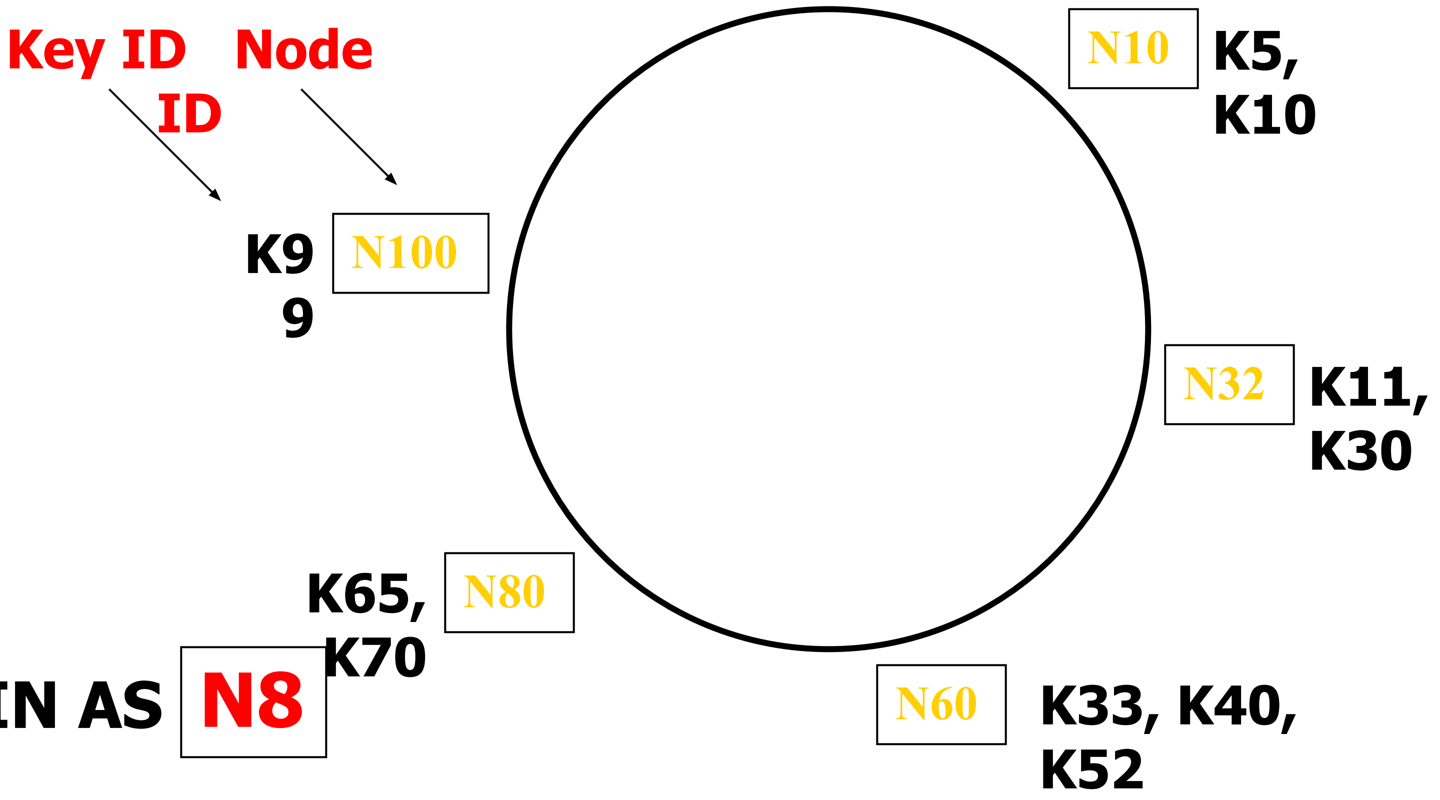
What's the cost of a node join?

How many nodes do I need to update (to correct routing tables)

How many messages do I need to send (to correct routing tables)

	Consistent Hashing	Distributed Hash Table
Lookup:	*O(1)*	*O(log N)*
Routing state per node:	*O(N)*	*O(log N)*
Node joins	*??*	

Consistent hashing

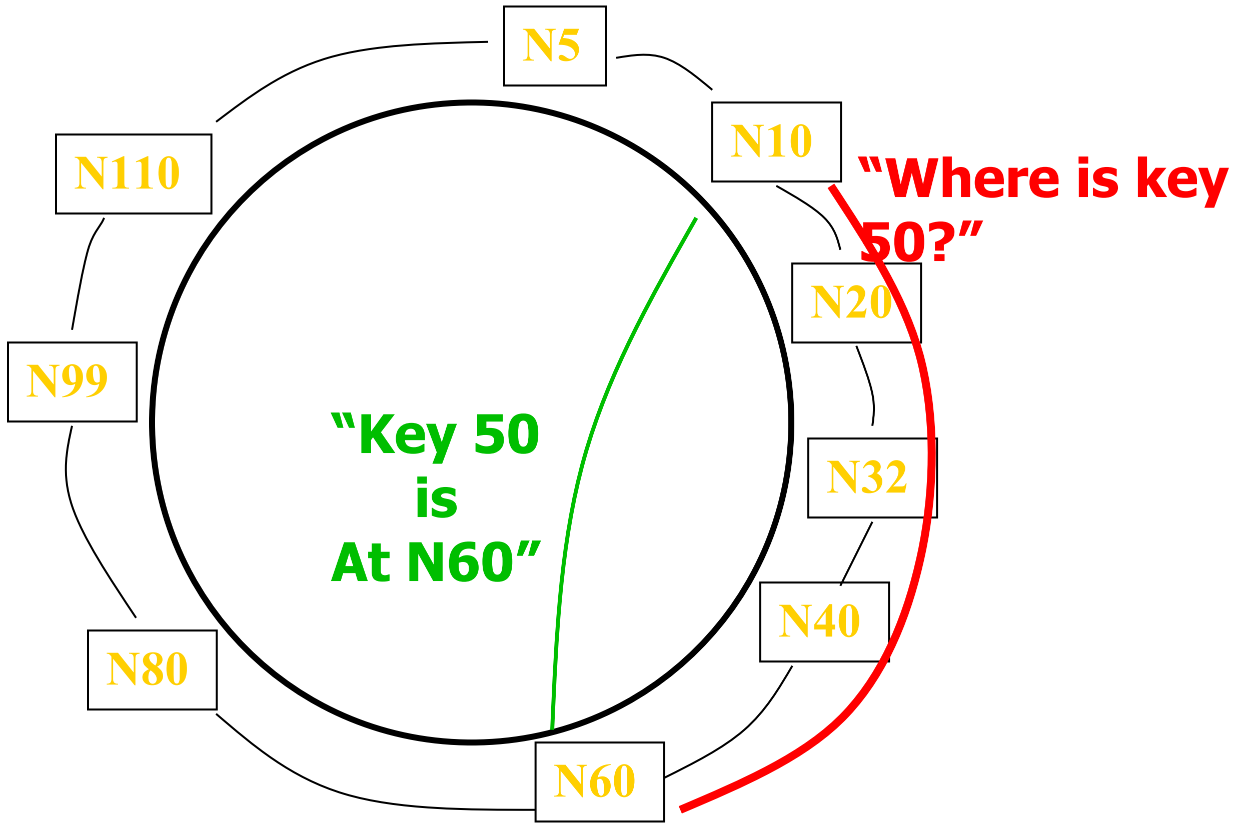


WILL JOIN AS $\boxed{N8}$

	Consistent Hashing	Distributed Hash Table
Lookup:	*O(1)*	*O(log N)*
Routing state per node:	*O(N)*	*O(log N)*
Node joins	*O(N)*	*???*

DHT: What info to maintain to route correctly?

- (at a minimum) Lookups correct if each node can maintain its successor. [invariant to maintain]
- Finger table: acceleration only, can be approximates



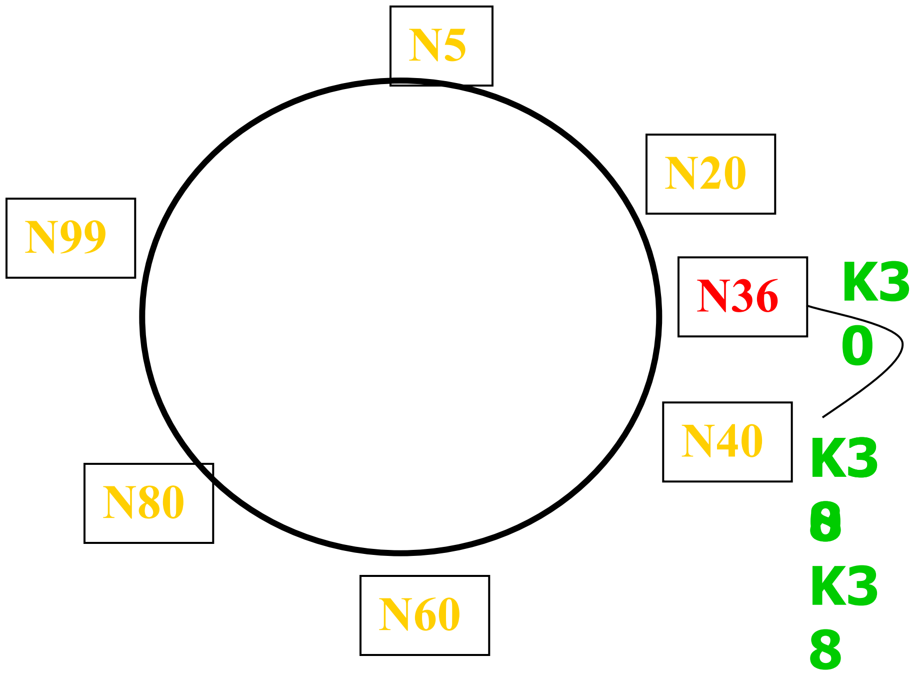
Joining the Ring: DHT

- Steps of the process [does order matter?]
  - Identify predecessor and successor nodes
  - [copy keys from successor to new node]
  - Announce yourself to predecessor and successor
  - Initialize 'fingers'/shortcuts of new node [can be done lazily]
  - Update fingers of existing nodes [can be done lazily]
  - [delete extra k/v pairs] [can be done lazily]

- Invariants to maintain to ensure correctness
  - Each node's maintains its successor
  - *successor(k)* is responsible for monitoring *k*

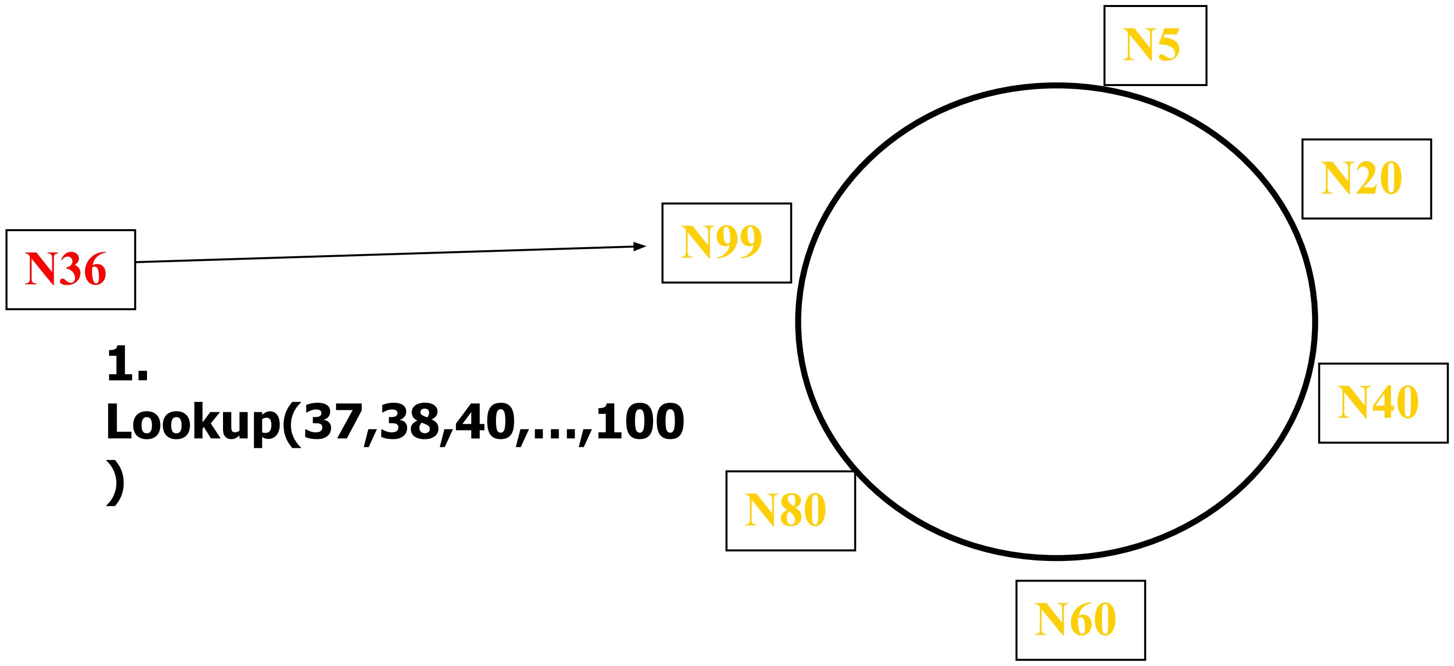
Join: Transfer Keys

- Only keys in the range are transferred



Copy keys 21..36 from N40 to N36

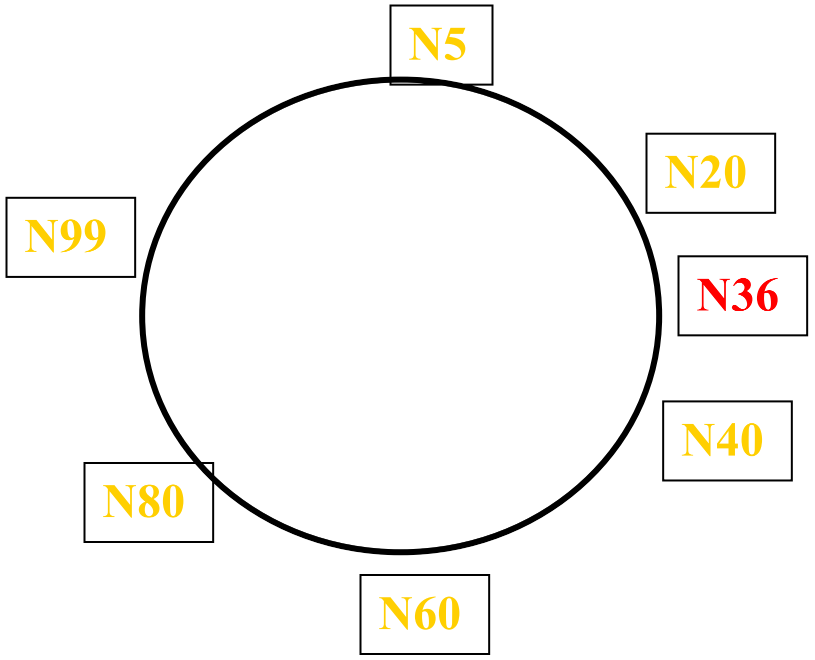
- How to initialize new node’s finger table?



Join: Update Fingers of Existing Nodes

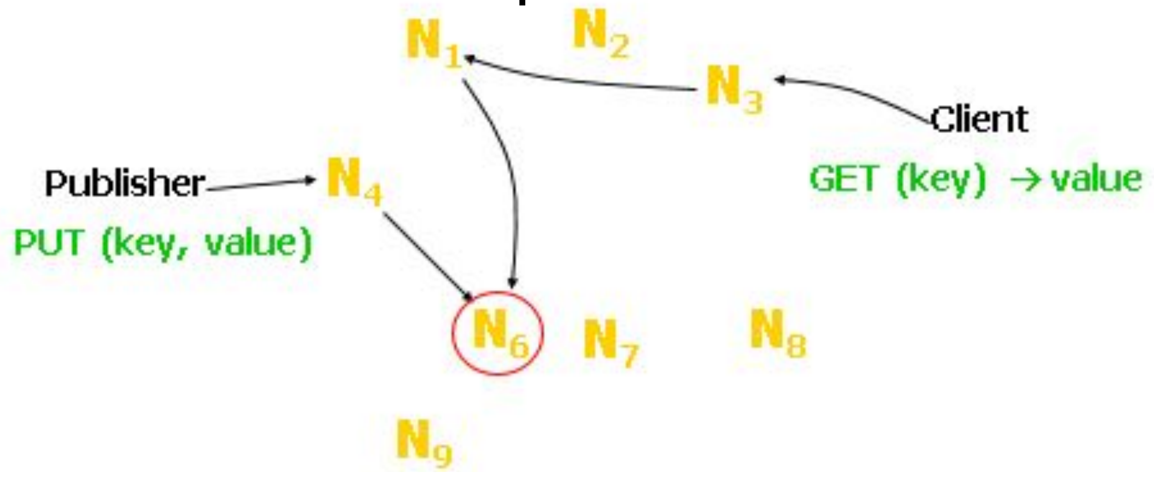
- New node calls update function on existing nodes
- Existing nodes recursively update fingers of other nodes

Note: updates can be lazy.



Design Issues

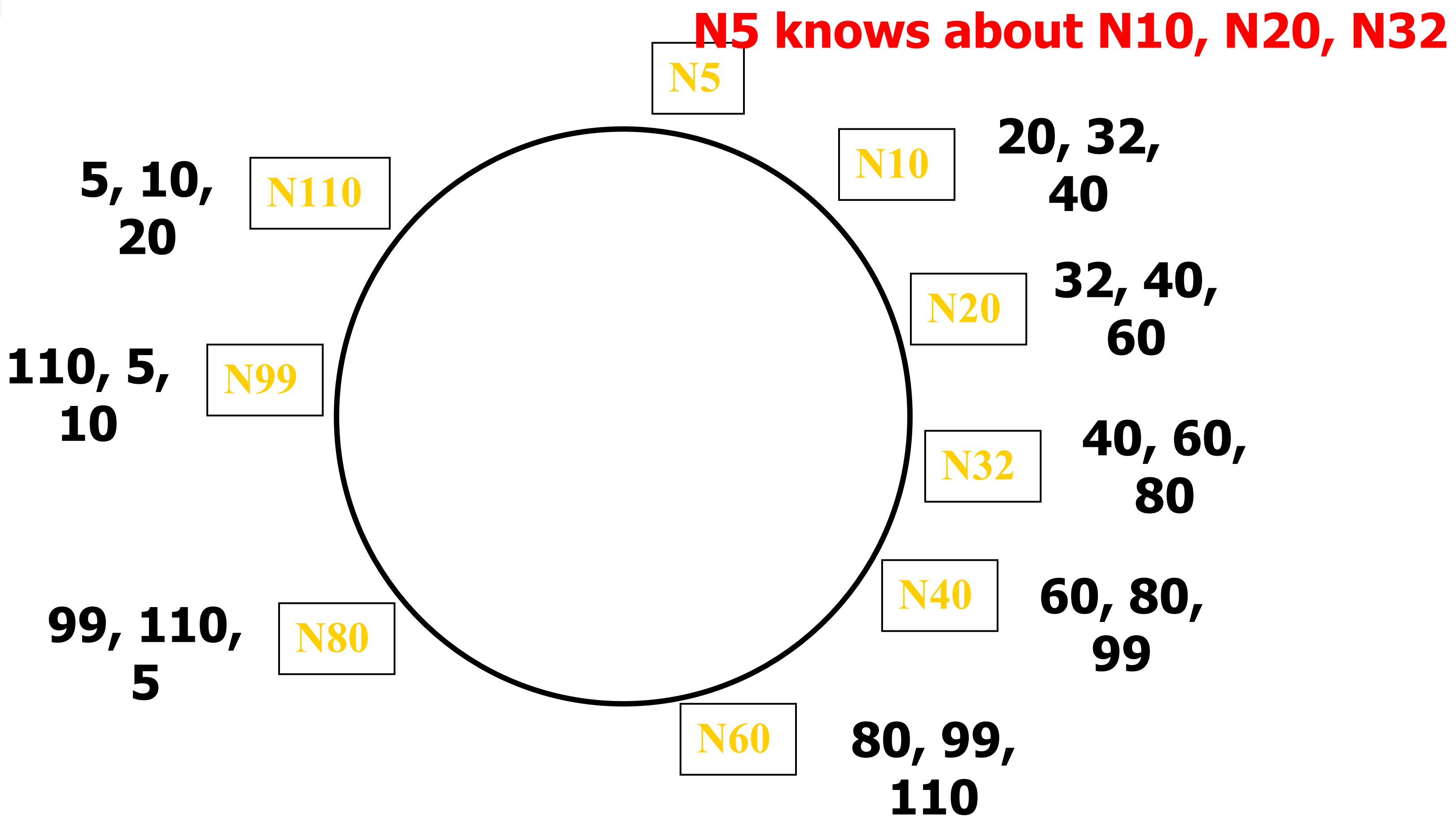
- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
  - [**(i) maintain routing structure**, (ii) maintain data]
- Issue 5: How to deal with node heterogeneity
- Issue 6: Iterative vs. recursive routing
- Issue 7: Performance optimizations



	Consistent Hashing	Distributed Hash Table
Lookup:	*O(1)*	*O(log N)*
Routing state per node:	*O(N)*	*O(log N)*
Node joins	*O(N)*	*O(log N)*
Node failure	*???*	

	Consistent Hashing	**Distributed Hash Table**
Lookup:	*O(1)*	*O(log N)*
Routing state per node:	*O(N)*	*O(log N)*
Node joins	*O(N)*	*O(log N)*
Node failure	*O(N)*	***????***

DHT Fault-tolerance: Successor Lists Ensure Robust Lookup



- Each node remembers ***R*** successors
- Lookup can skip over dead nodes

How does one dimension the successor list?

What is the chance that the system works correctly after N nodes fail?

system fails if at least one has lost all its successors (the ring can not be repaired)

- ***F*** – the fraction of the nodes that fail
- ***R*** – *length of successor list ;*
- ***N*** – *nodes in the system*

$$P(\textit{all successors of a specific node have failed}) = F^R$$

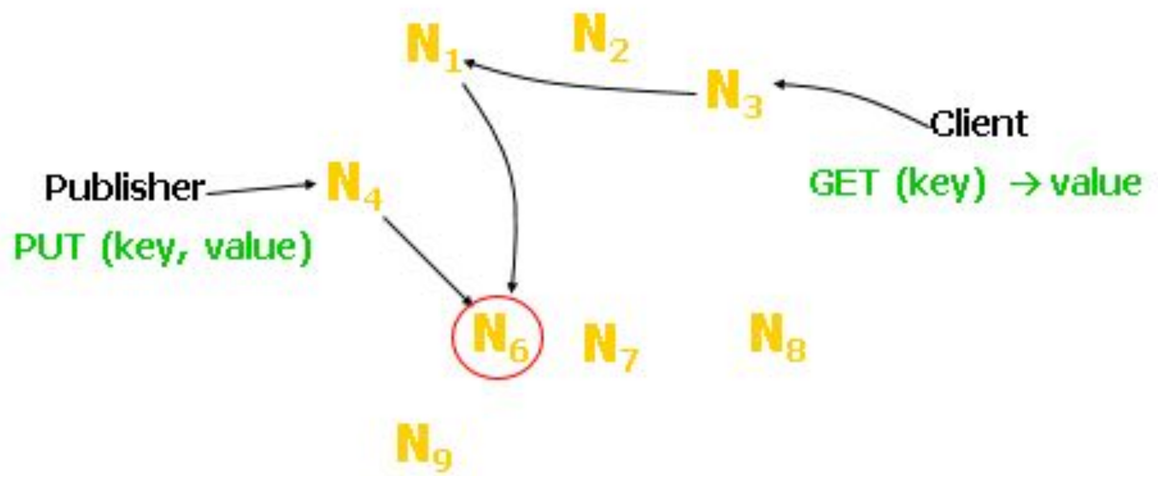
$$P(\textit{no system failure}) =$$

$$= P(\textit{all nodes are ok}) =$$

$$= (1-F^R)^N$$

Design Issues

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
  - [(i) maintain routing structure, **(ii) maintain data**]
- Issue 5: How to deal with node heterogeneity

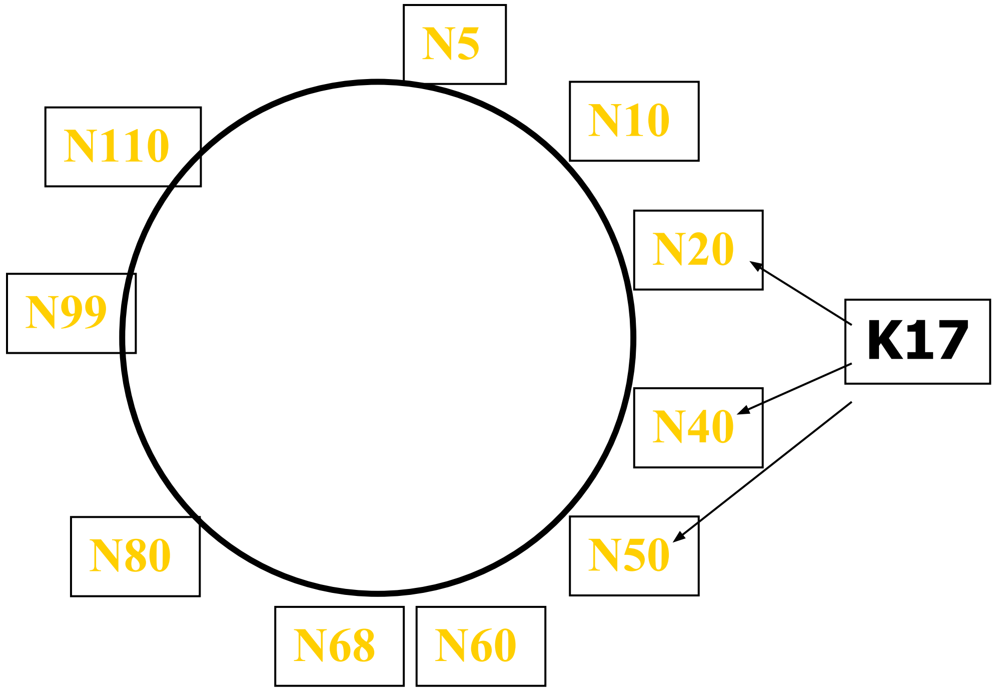


How to maintain data?

The same solutions work for consistent hashing and DHT

- Nodes failure: replicate data
  - Pick an uniform choice of nodes to do replication
    - E.g., replicate to R successors
- Node joins: migrate data
  - (Lazily) delete unnecessary replicas

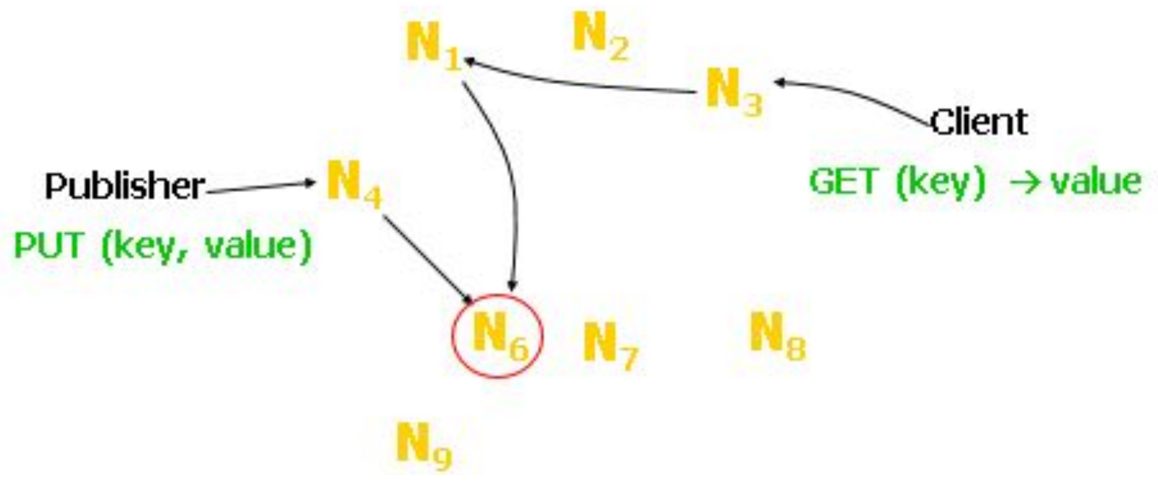
Replicates [k,v] pairs at *R* Successors



- Replicas are easy to find if successor fails
- Hashed node IDs ensure independent failure

Design Issues

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity
- Issue 6: Iterative vs. recursive routing
- Issue 7: Performance optimizations



Problem: How to load balance when nodes are heterogeneous?

Solution idea: Each node owns an ID space proportional to its 'power'

Virtual Nodes:

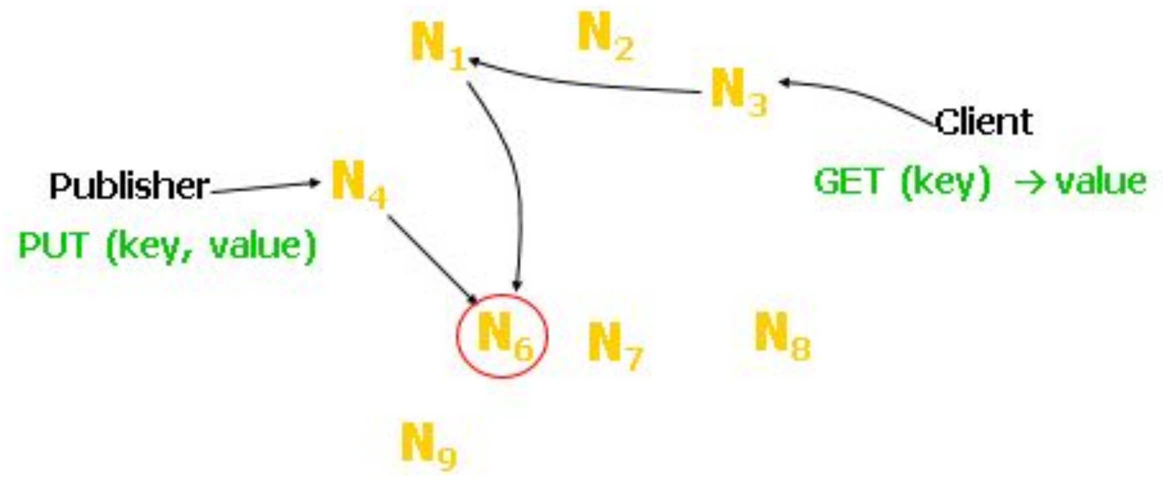
- Each physical node is responsible for multiple (similar) virtual nodes.
- Virtual nodes are treated the same

Advantages: load balancing, incremental scalability,

- Dealing with heterogeneity: The number of virtual nodes that a node is responsible for can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.
- When a node joins (if it supports many VN) it accepts a roughly equivalent amount of load from each of the other existing nodes.
- If a node becomes unavailable the load handled by this node is evenly dispersed across the remaining available nodes.

Design Issues

- Issue 1: How to map keys to nodes?
- Issue 2: How to route requests?
- Issue 3: How to deal with node joins?
- Issue 4: How to deal with node failures?
- Issue 5: How to deal with node heterogeneity
- Issue 6: Iterative vs. recursive routing
- Issue 7: Performance optimizations [link]



- Design Choices:
  - Routing (Recursive vs. Iterative).
  - At-most-once cache? (Front-End vs. Back-End)

Recap: Properties

- Decentralized / symmetric design: nodes have similar roles
- Scalable: multiple axes
  - network traffic overhead, state at nodes, routing cost, …
- Incremental scalability
- Efficient: find items quickly (latency)
- Dynamic: deals with node failure, join
- General-purpose: flat naming, heterogeneous platform

BACKUP

Idea: Key = Hash(Value)

- Why a ‘secure’ hash function?
  - One way
  - Uniform distribution for hash(x)

- Some games played:

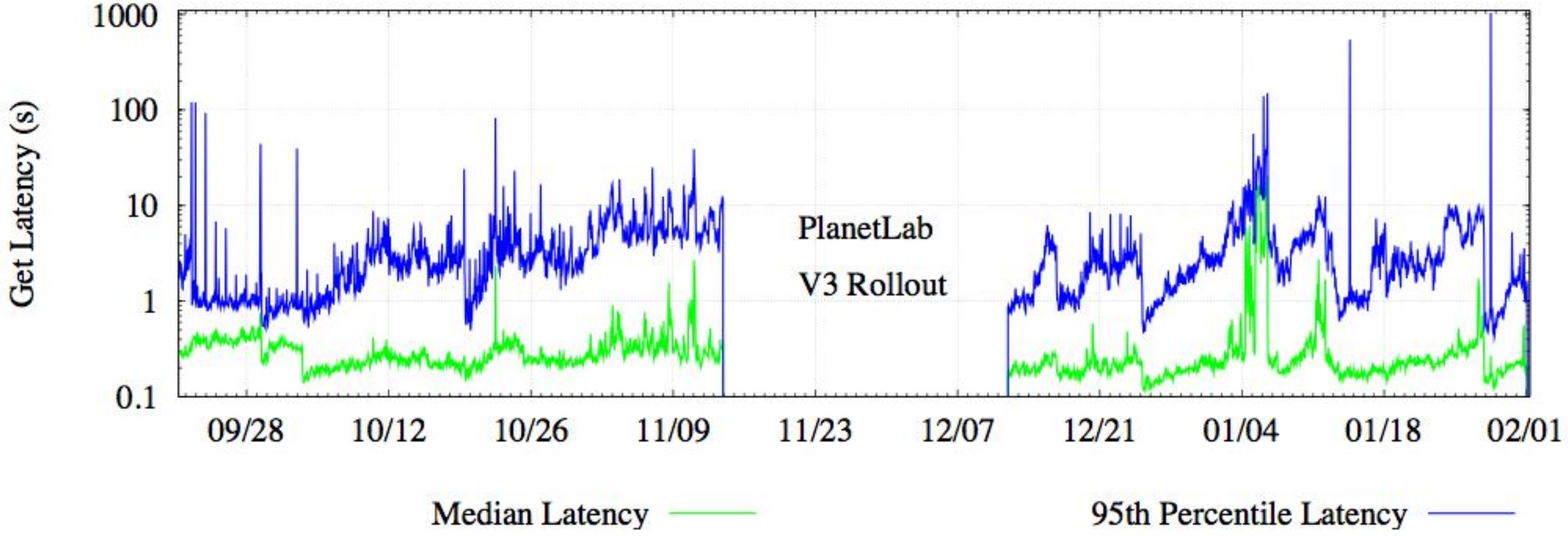
  Client uses: key = hash(value)

  PUT (hash(value), value)

  Advantages
  - Free error detection
  - Attacker can not change value (as long as client remembers the key)

Some experimental results and performance optimizations

Fixing the Embarrassing Slowness of OpenDHT on PlanetLab (2005)



- Median RTT between hosts ~ 140 ms
- Median get performance: 200 ms
- 95th percentile get latency is high!
- Generally measured in *seconds*
- And even median spikes up from time to time

Delay-Aware Routing

	Latency (ms)		Cost	
Mode	50th	99th	Msgs	Bytes
Greedy	150	4400	5.5	1800
Delay-Aware	100	1800	6.0	2000

- Latency drops by 30-60%
- Cost goes up by only ~10%