Sharding

Scaling Paxos: Shards

We can use Paxos to decide on the order of operations, e.g., to a key-value store

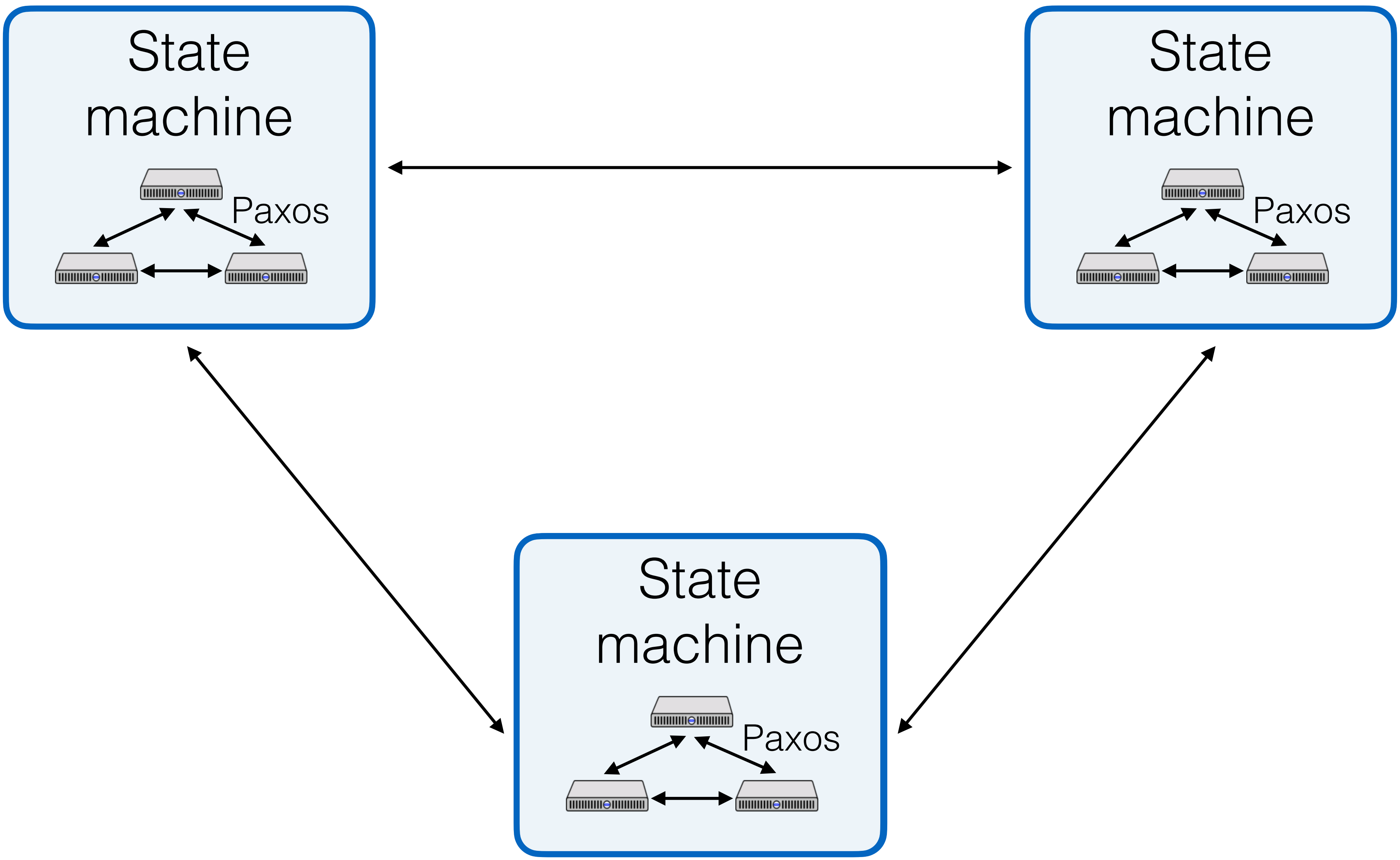
- leader sends each op to all servers
- practical limit on how ops/second

What if we want to scale to more clients?

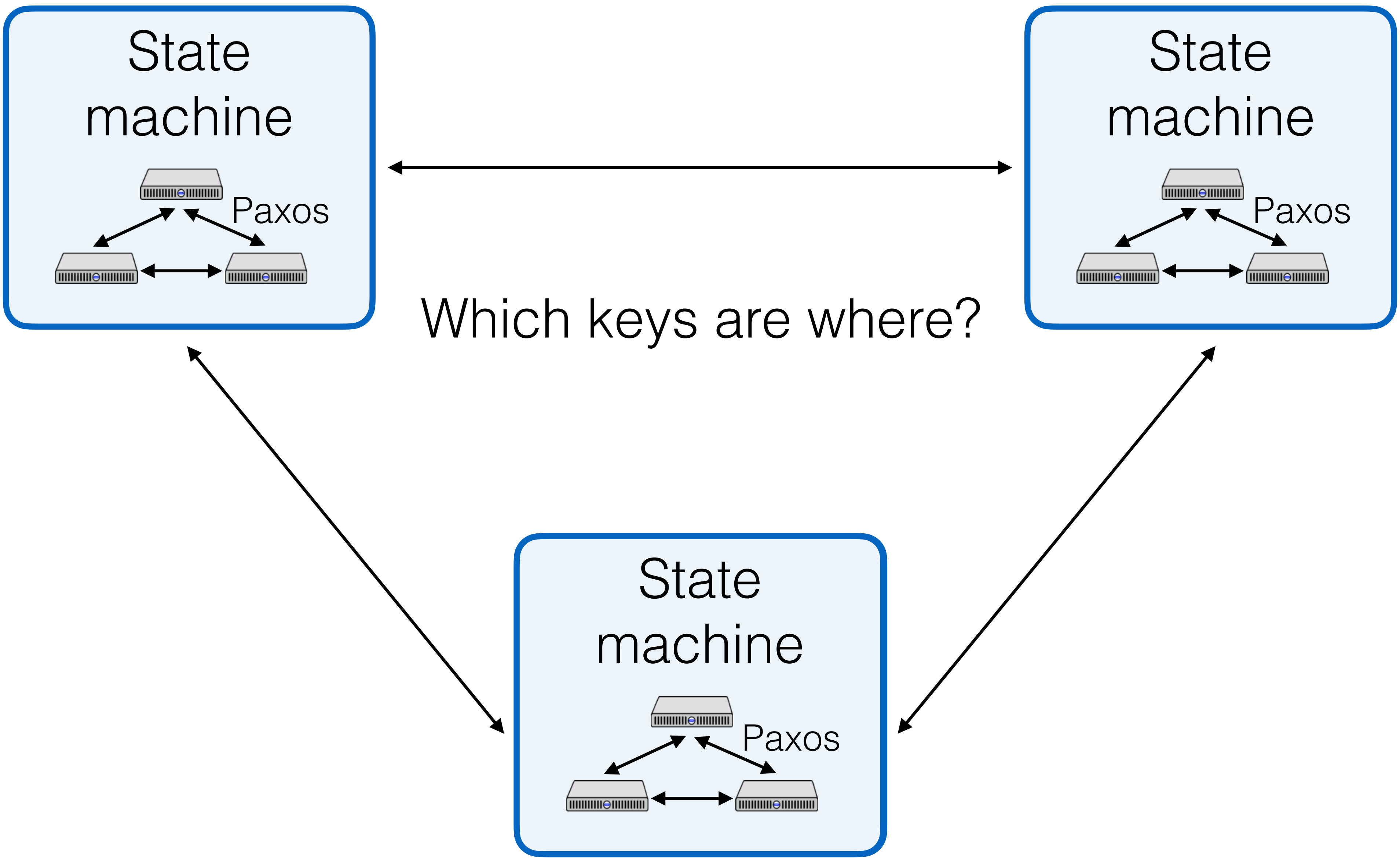
Sharding among multiple Paxos groups

- partition key-space among groups
- for single key operations, still linearizable

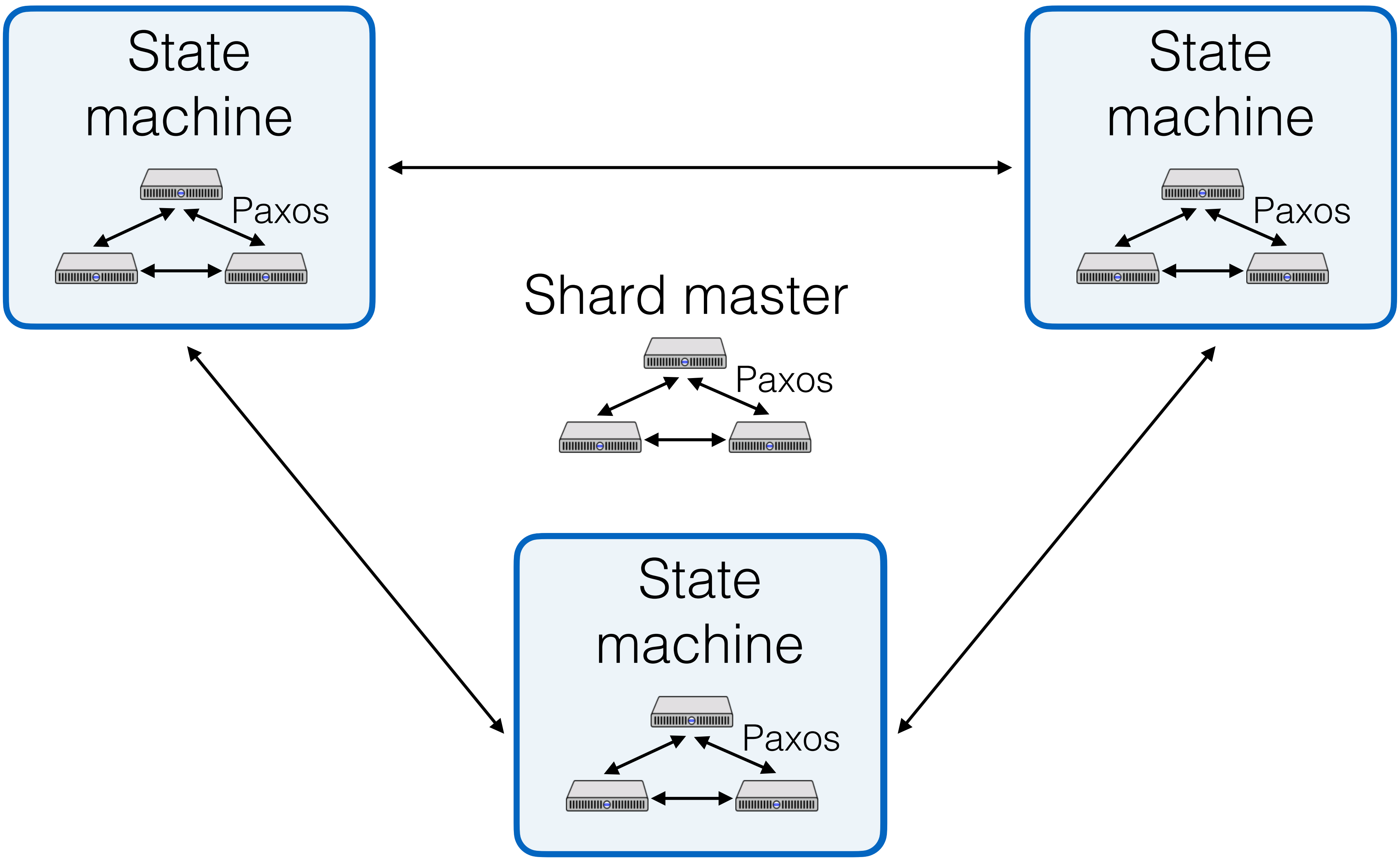
Replicated, Sharded Database



Replicated, Sharded Database



Lab 4 (and other systems)



Replicated, Sharded Database

Shard master decides

- which Paxos group has which keys

Shards operate independently

How do clients know who has what keys?

- Ask shard master? Becomes the bottleneck!
- Avoid shard master communication if possible

Can clients predict which group has which keys?

Recurring Problem

Client needs to access some resource

Sharded for scalability

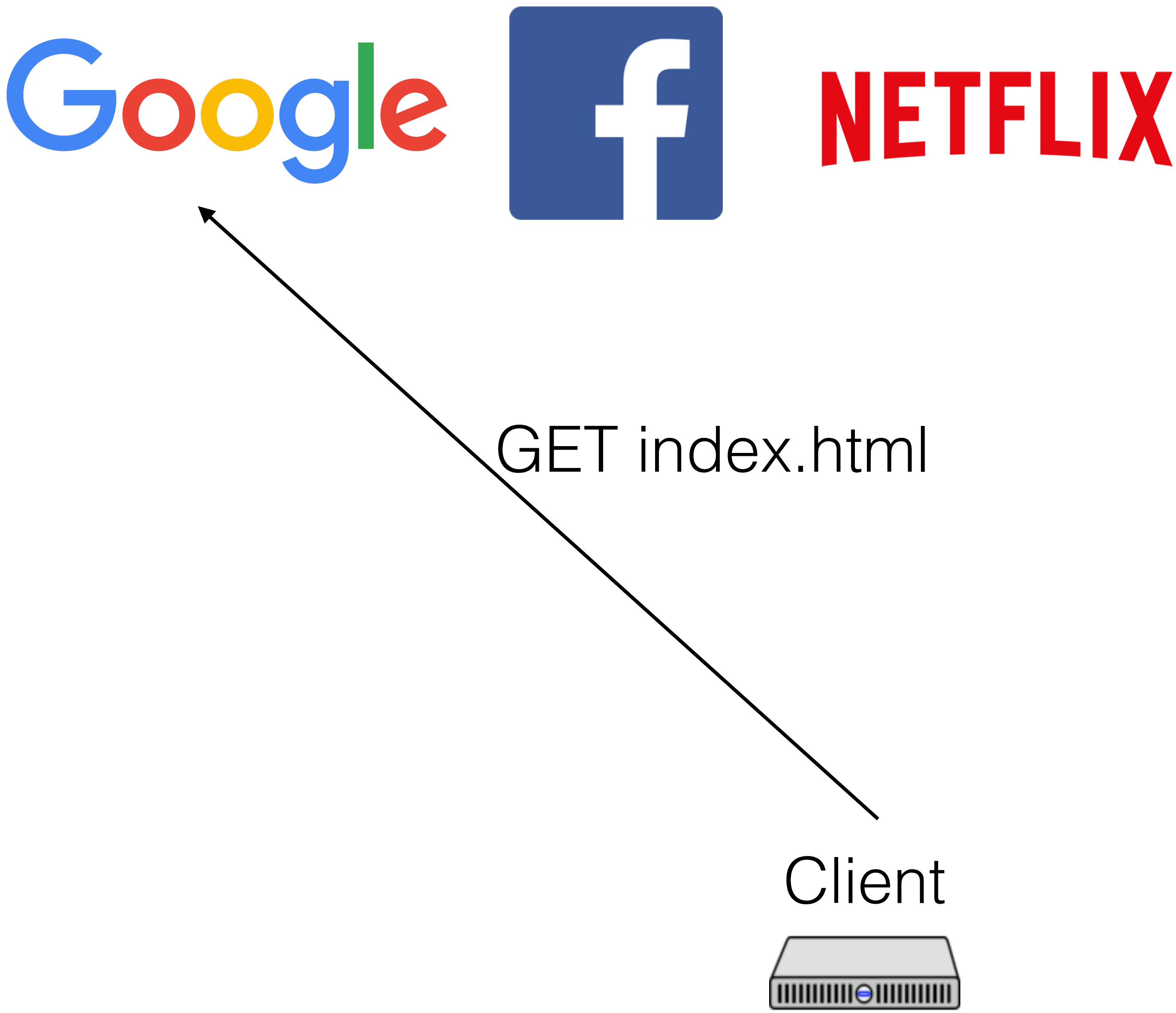
How does client find specific server to use?

Central redirection won't scale!

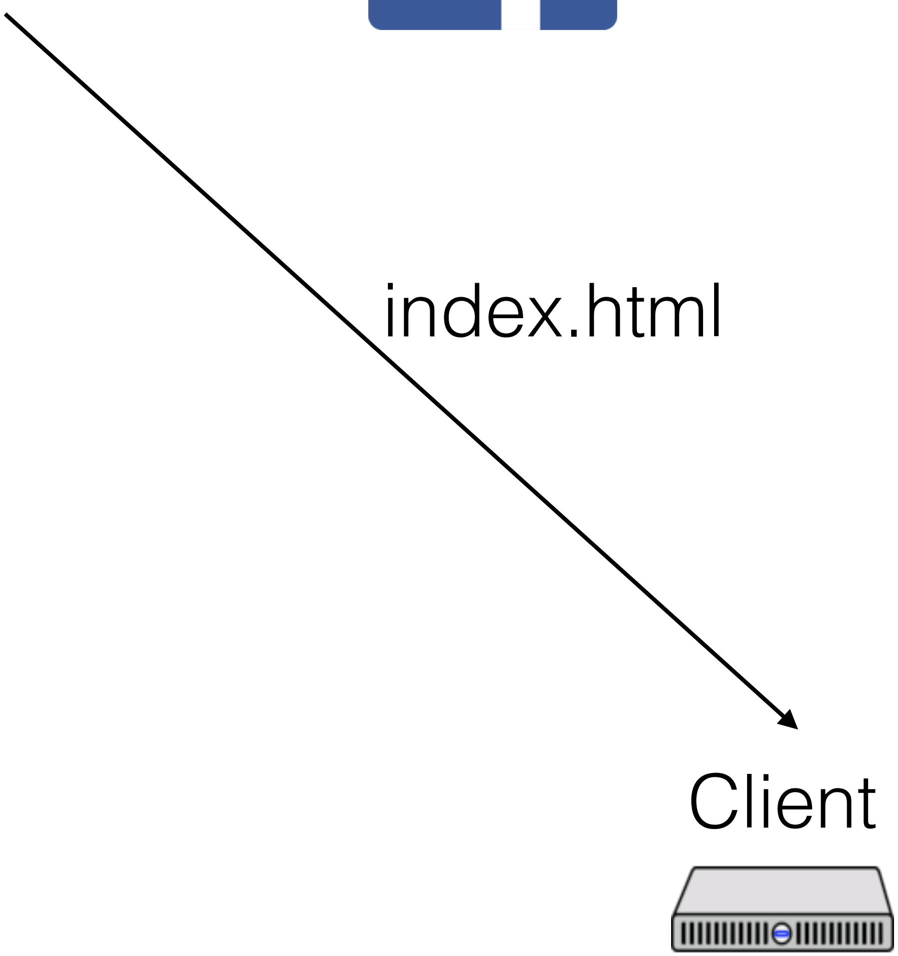
Another scenario

Client

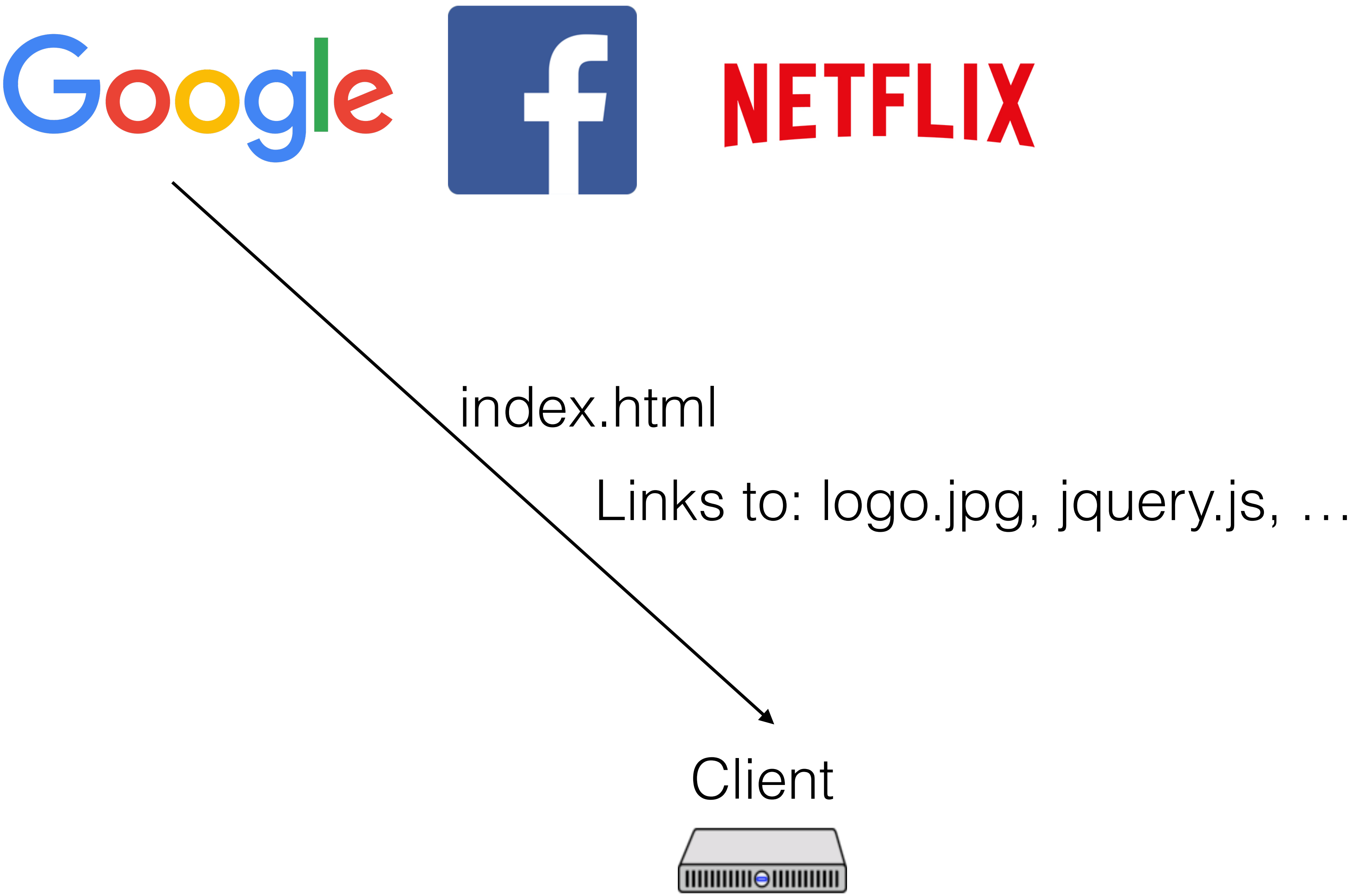
Another scenario



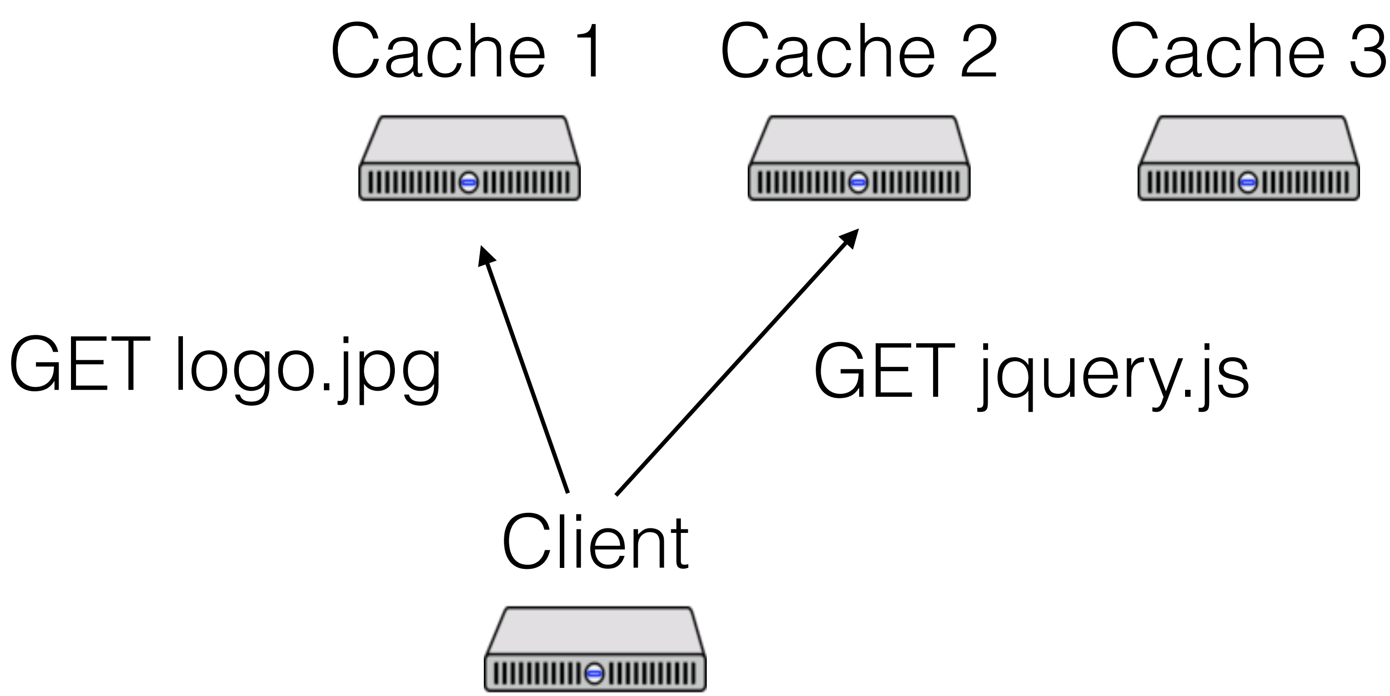
Another scenario



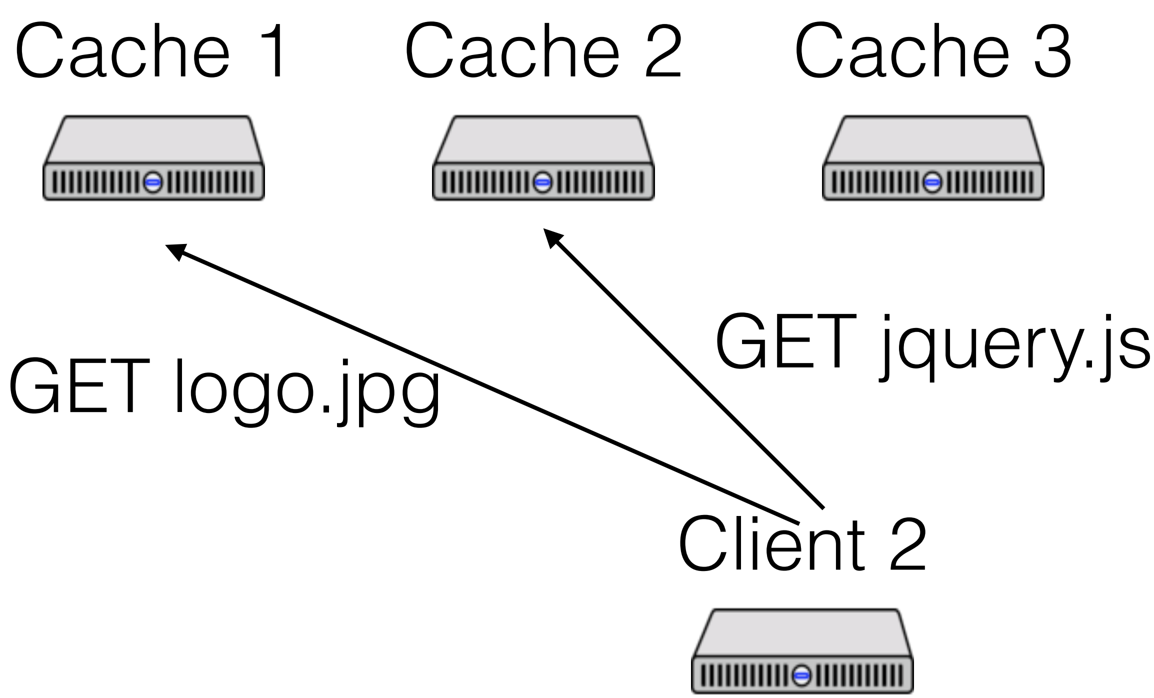
Another scenario



Another scenario



Another scenario



Other Examples

Scalable stateless web front ends (FE)

- cache efficient iff same client goes to same FE

Scalable shopping cart service

Scalable email service

Scalable cache layer (Memcache)

Scalable network path allocation

Scalable network function virtualization (NFV)

…

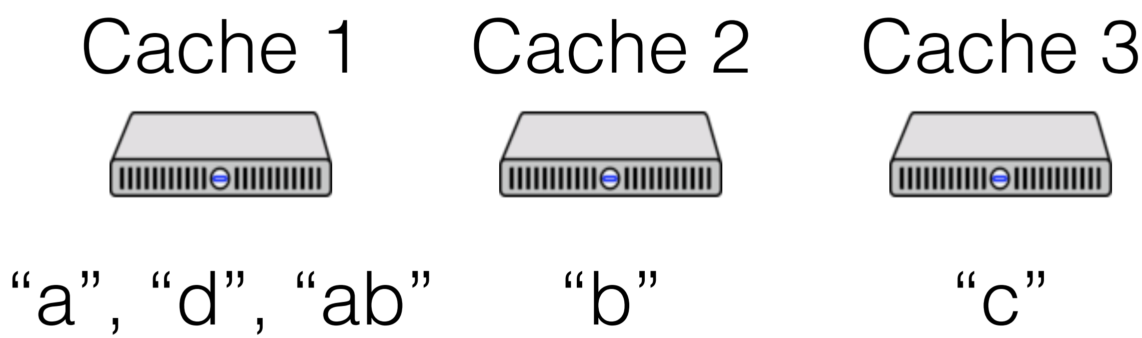
What’s in common?

Want to assign keys to servers with minimal communication, fast lookup

Requirement 1: clients all have same assignment

Proposal 1

For *n* nodes, a key *k* goes to *k* mod *n*



Proposal 1

For n nodes, a key k goes to $k \bmod n$

Cache 1 Cache 2 Cache 3

"a", "d", "ab" "b" "c"

Problems with this approach?

Proposal 1

For *n* nodes, a key *k* goes to *k* mod *n*

Problems with this approach?

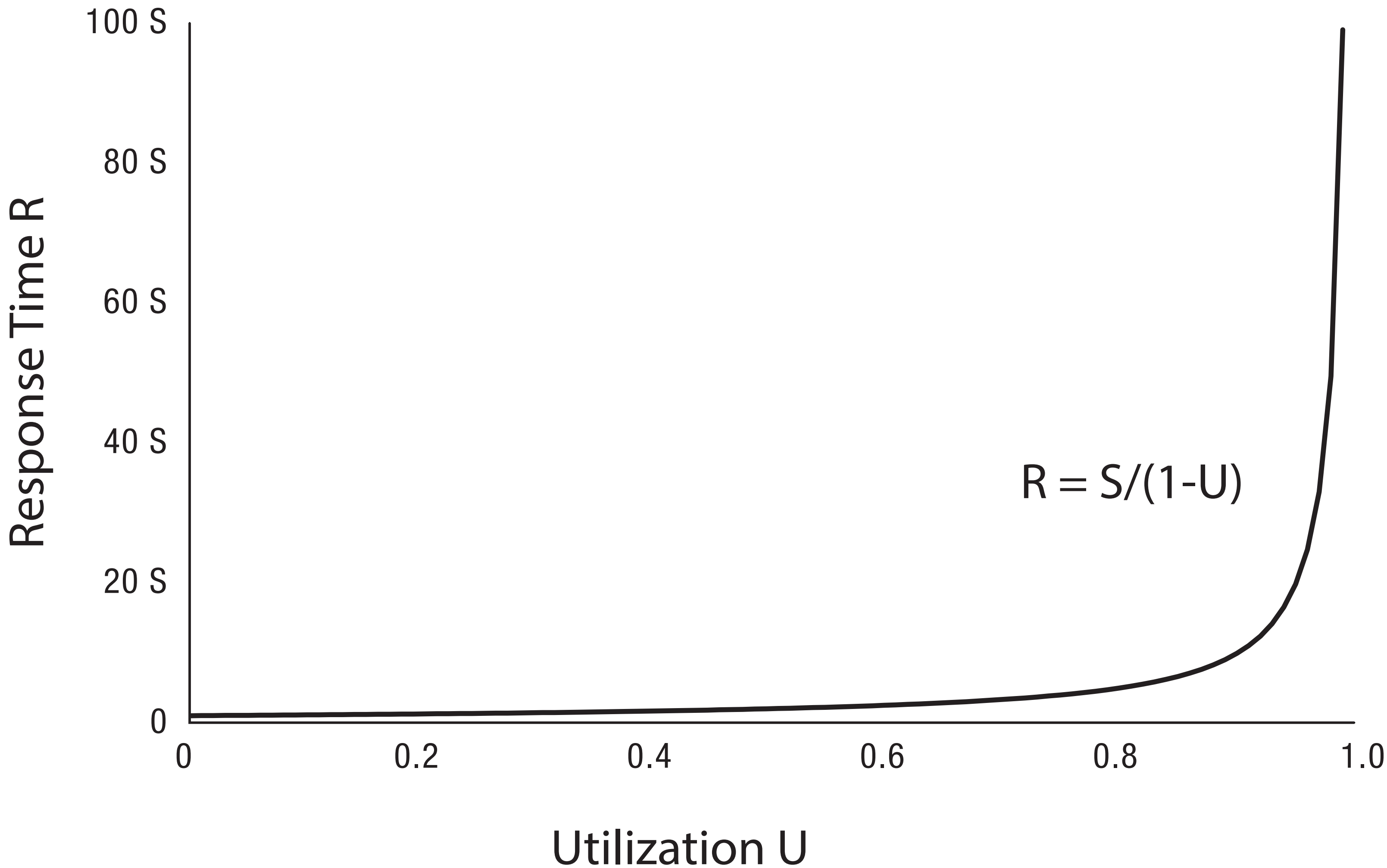
- uneven distribution of keys

A Bit of Queueing Theory

Assume Poisson arrivals:

- random, uncorrelated, memoryless
- utilization (U): fraction of time server is busy (0 - 1)
- service time (S): average time per request

Queueing Theory



Variance in response time ~ S/(1-U)^2

Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Proposal 2: Hashing

For *n* nodes, a key *k* goes to *hash(k)* mod *n*

Cache 1 Cache 2 Cache 3

h("a")=1 *h*("abc")=2 *h*("b")=3

Hash distributes keys uniformly

Proposal 2: Hashing

For *n* nodes, a key *k* goes to *hash(k)* mod *n*

Cache 1 Cache 2 Cache 3

h("a")=1 *h*("abc")=2 *h*("b")=3

Hash distributes keys uniformly

But, new problem: what if we add a node?

Proposal 2: Hashing

For *n* nodes, a key *k* goes to *hash(k)* mod *n*

Cache 1 Cache 2 Cache 3 Cache 4

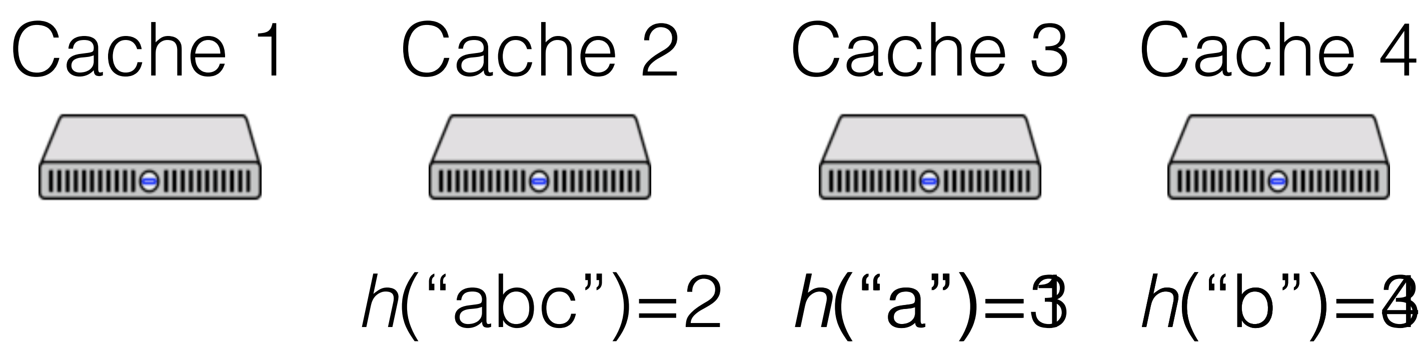
h(“a”)=1 *h*(“abc”)=2 *h*(“b”)=3

Hash distributes keys uniformly

But, new problem: what if we add a node?

Proposal 2: Hashing

For *n* nodes, a key *k* goes to *hash(k)* mod *n*



Hash distributes keys uniformly

But, new problem: what if we add a node?

Proposal 2: Hashing

For *n* nodes, a key *k* goes to *hash(k)* mod *n*

Cache 1 Cache 2 Cache 3 Cache 4

h("abc")=2 *h*("a")=3 *h*("b")=4

Hash distributes keys uniformly

But, new problem: what if we add a node?

- Redistribute a lot of keys! (on average, all but K/n)

Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

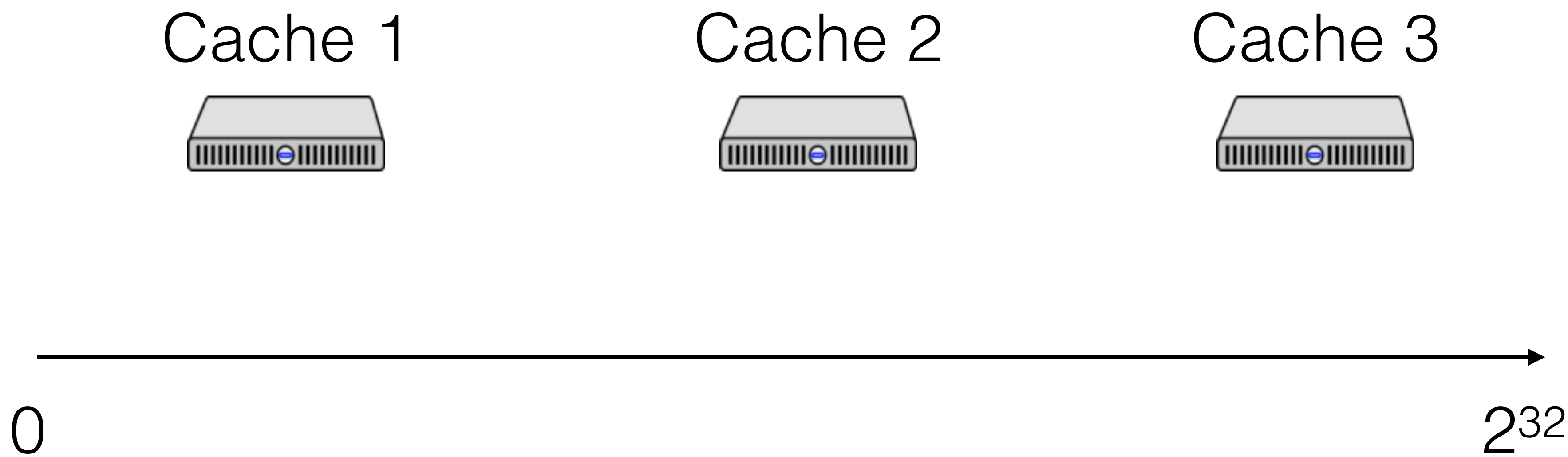
Requirement 3: add/remove node moves only a few keys

Proposal 3: Consistent Hashing

First, hash the node ids

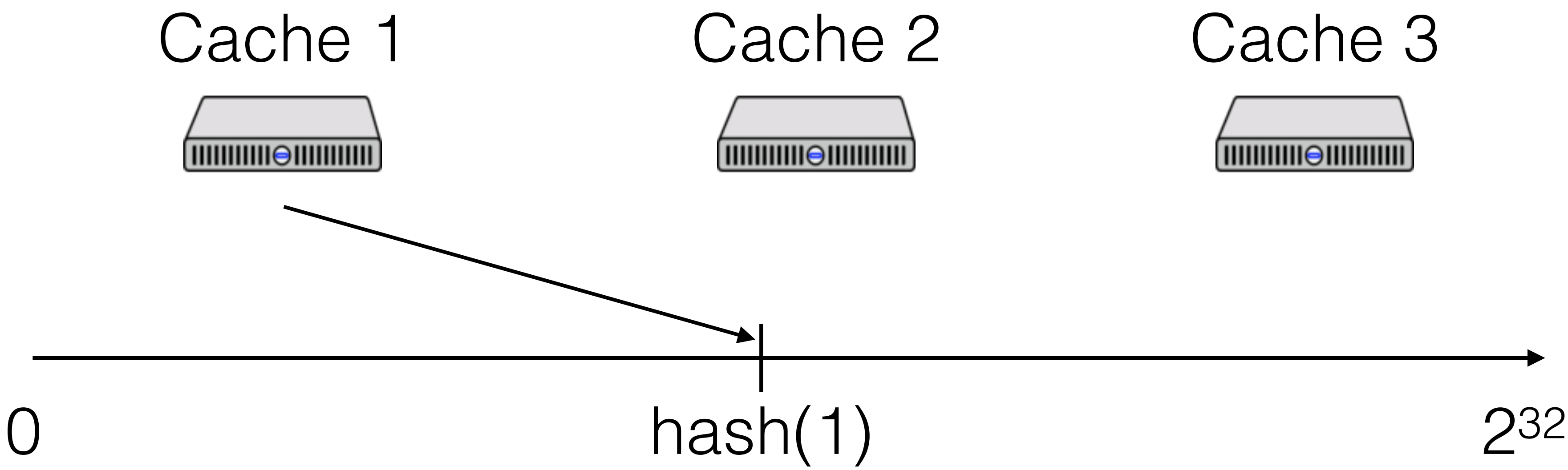
Proposal 3: Consistent Hashing

First, hash the node ids



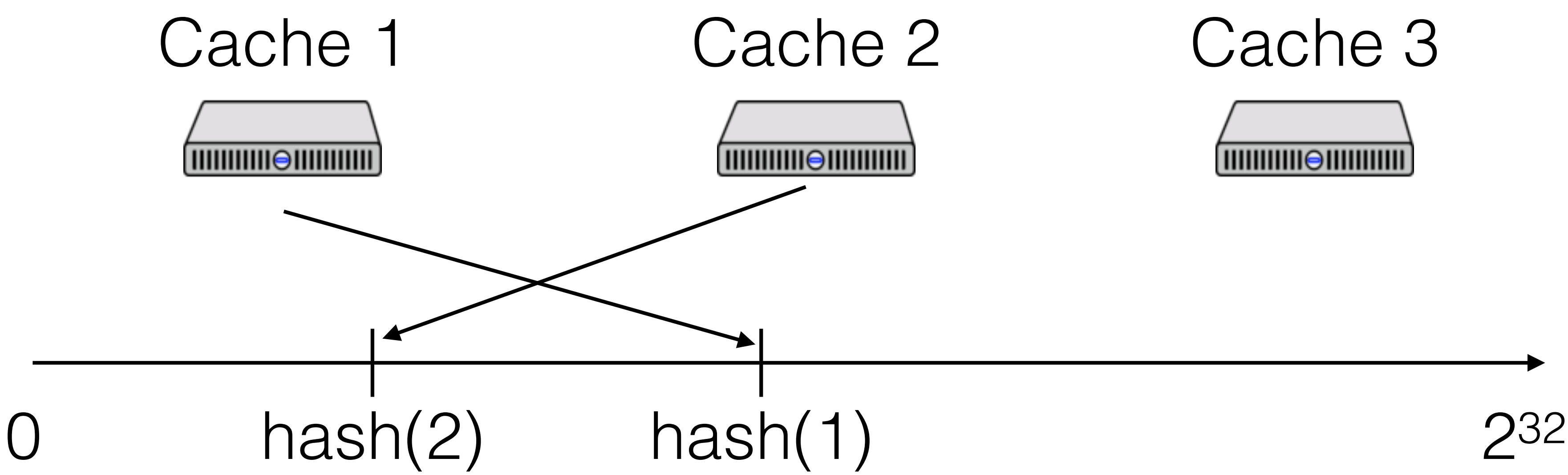
Proposal 3: Consistent Hashing

First, hash the node ids



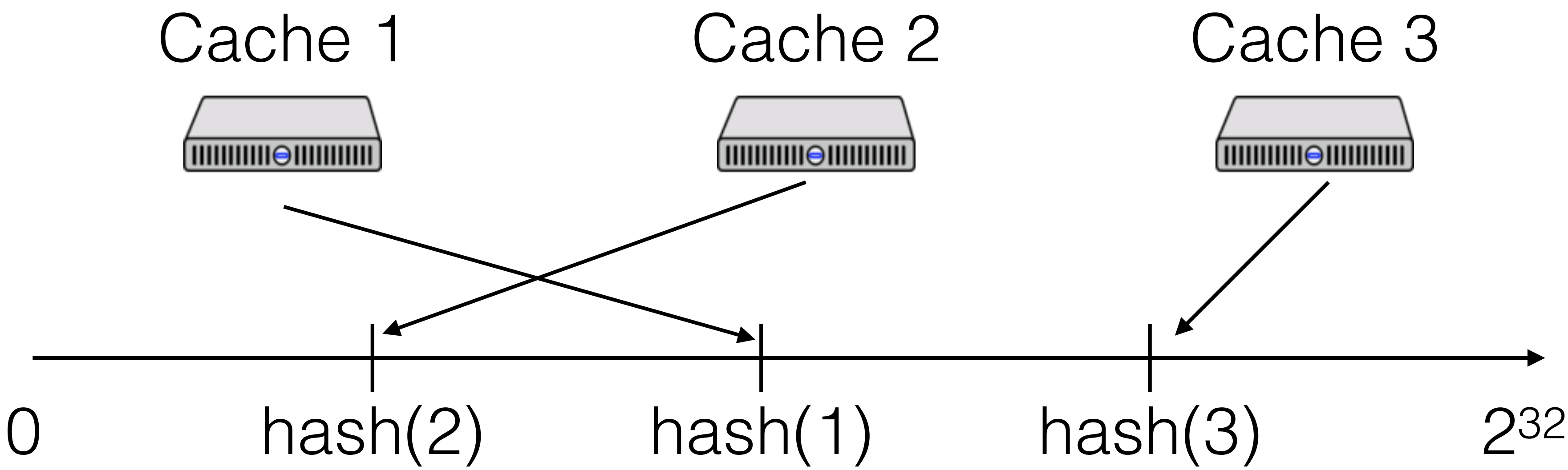
Proposal 3: Consistent Hashing

First, hash the node ids



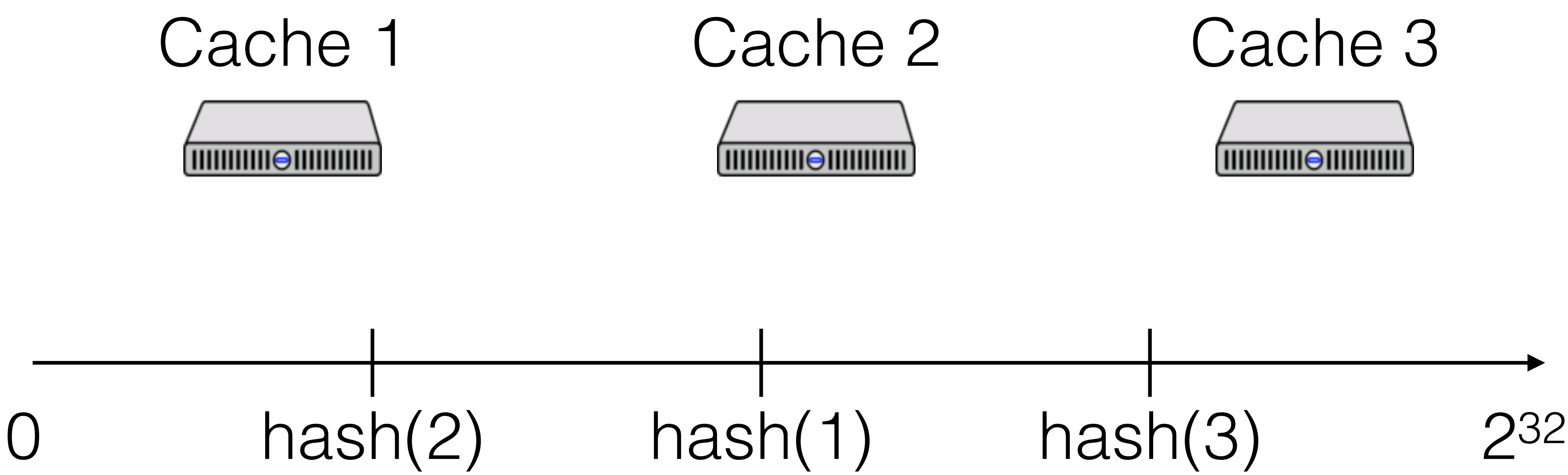
Proposal 3: Consistent Hashing

First, hash the node ids



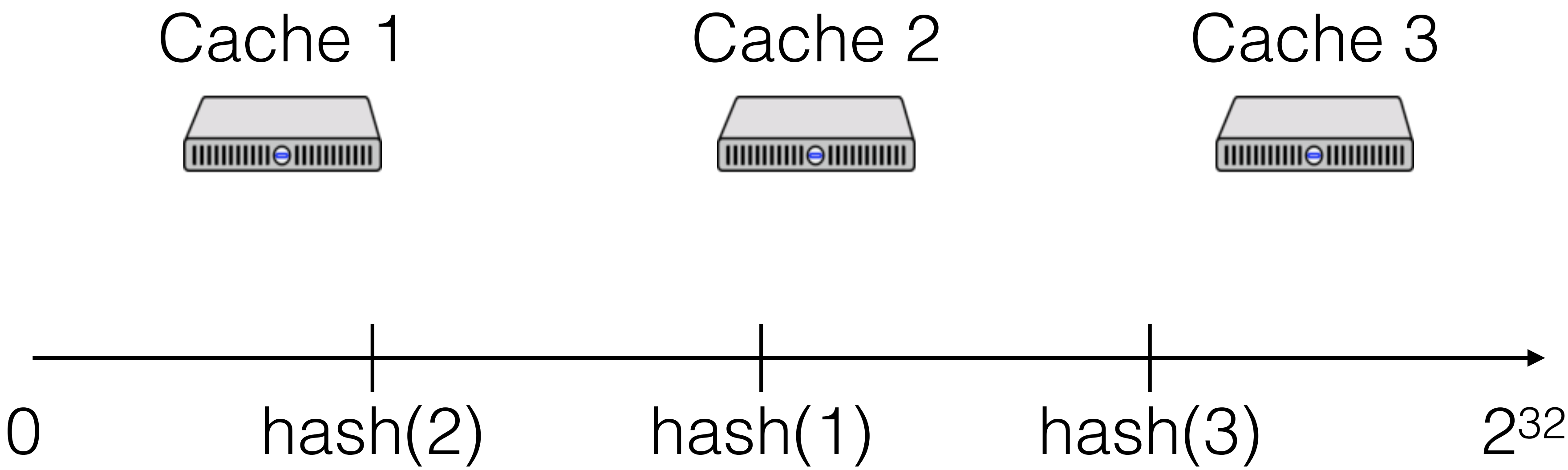
Proposal 3: Consistent Hashing

First, hash the node ids



Proposal 3: Consistent Hashing

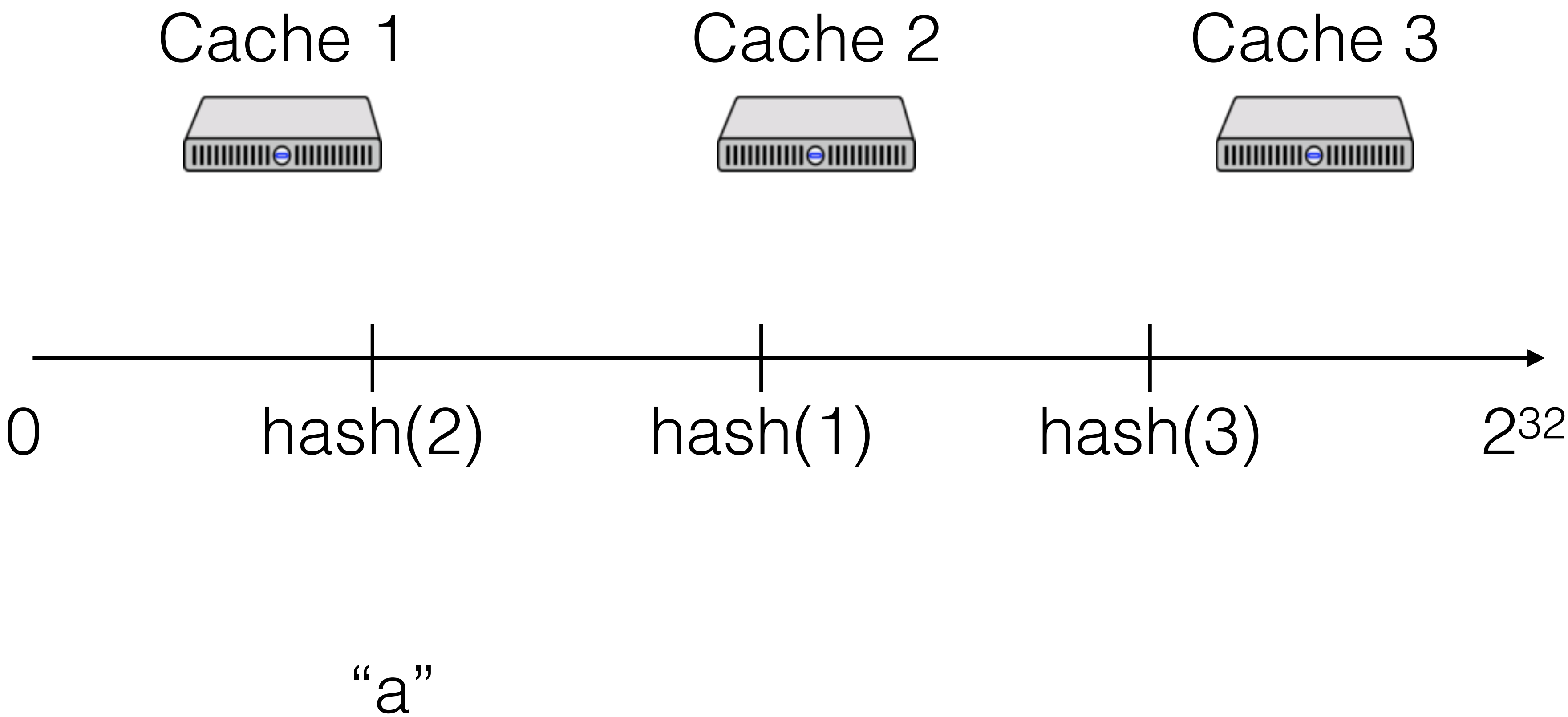
First, hash the node ids



Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

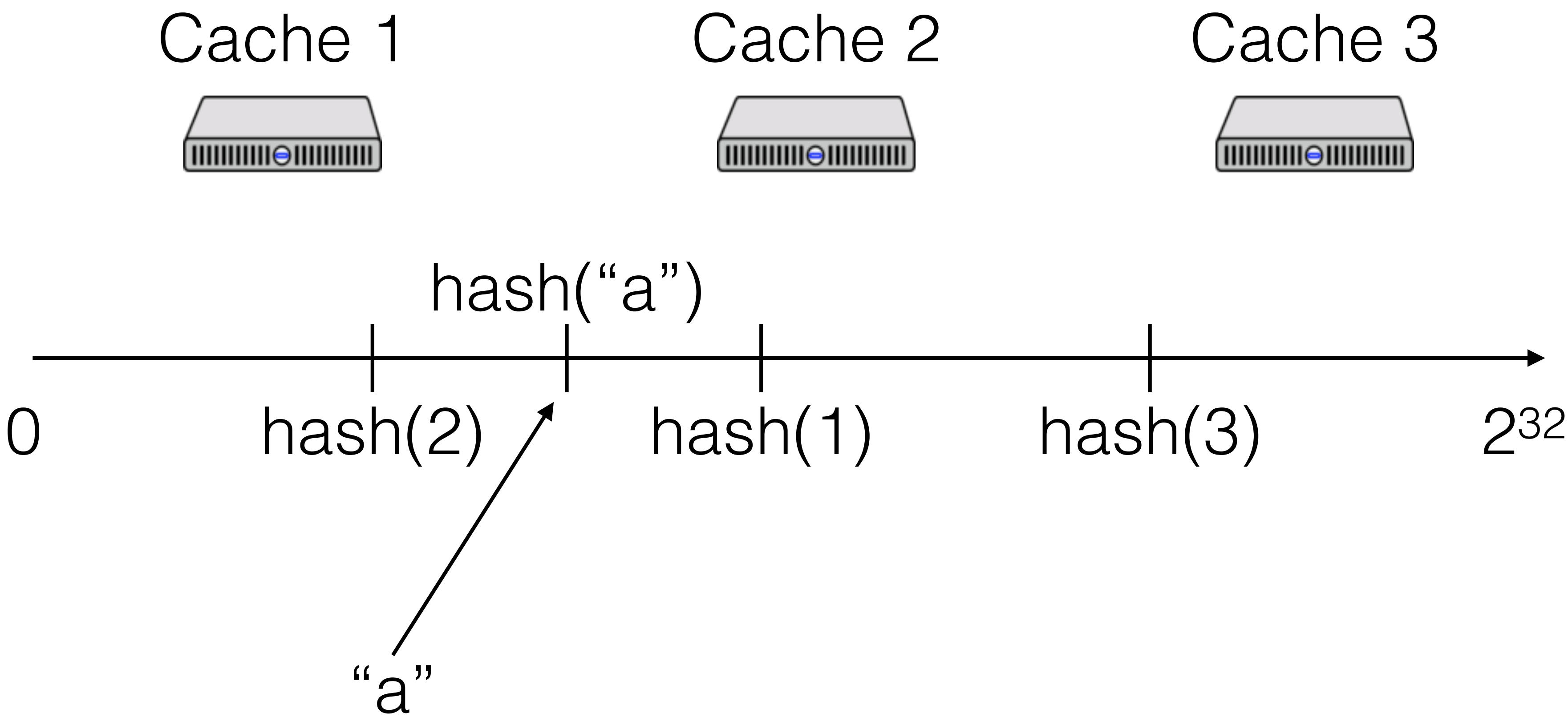
First, hash the node ids



Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

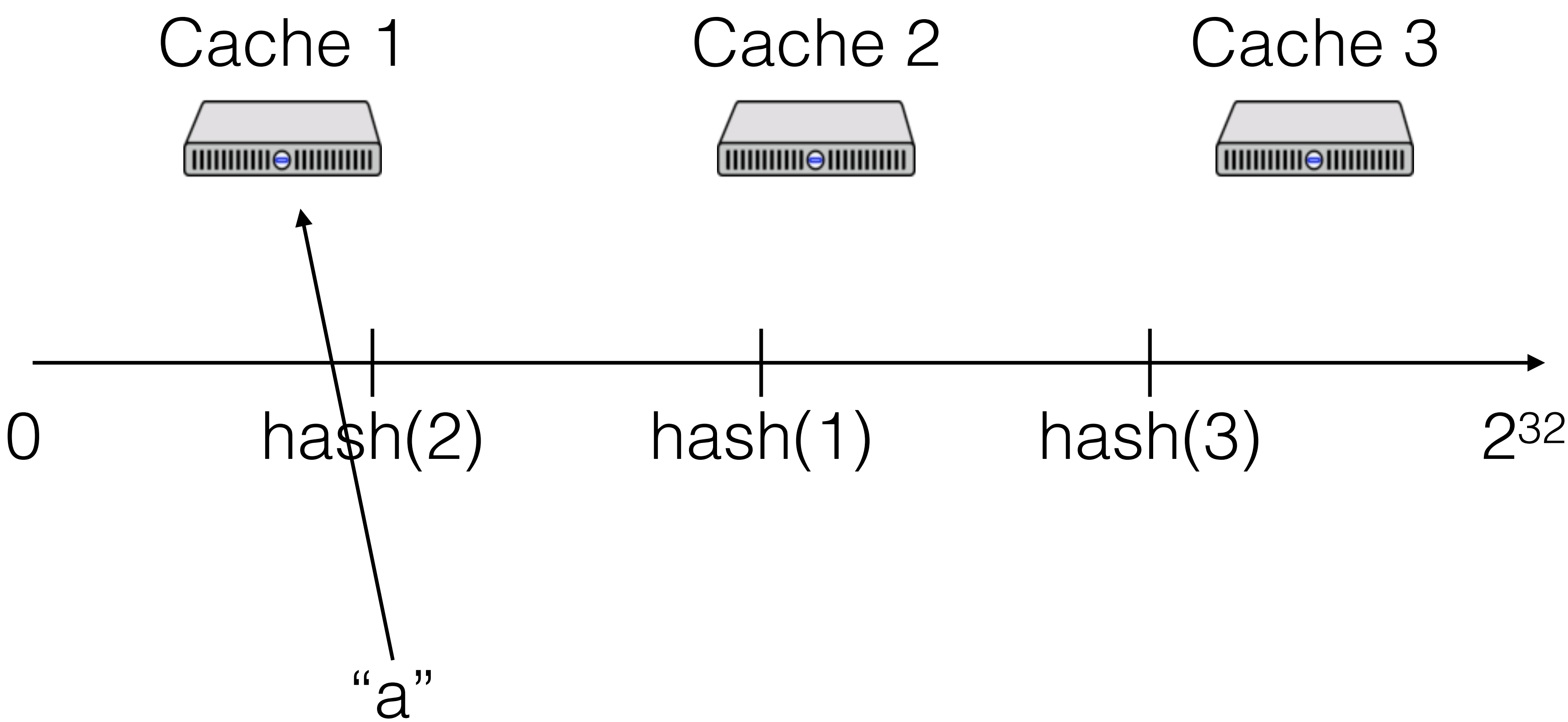
First, hash the node ids



Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

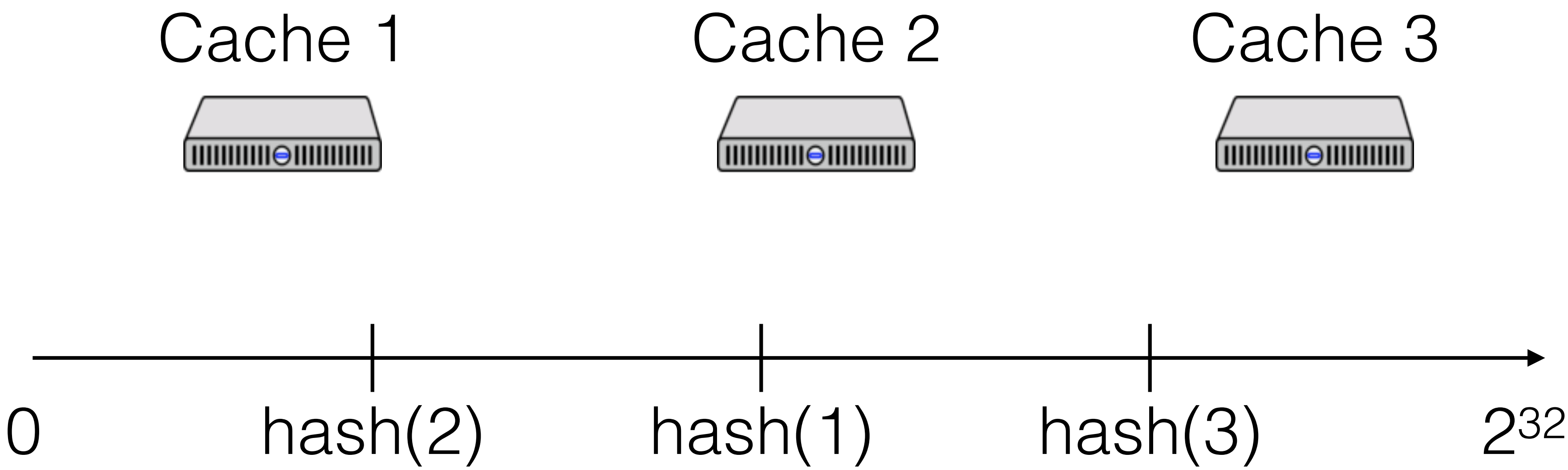
First, hash the node ids



Keys are hashed, go to the "next" node

Proposal 3: Consistent Hashing

First, hash the node ids

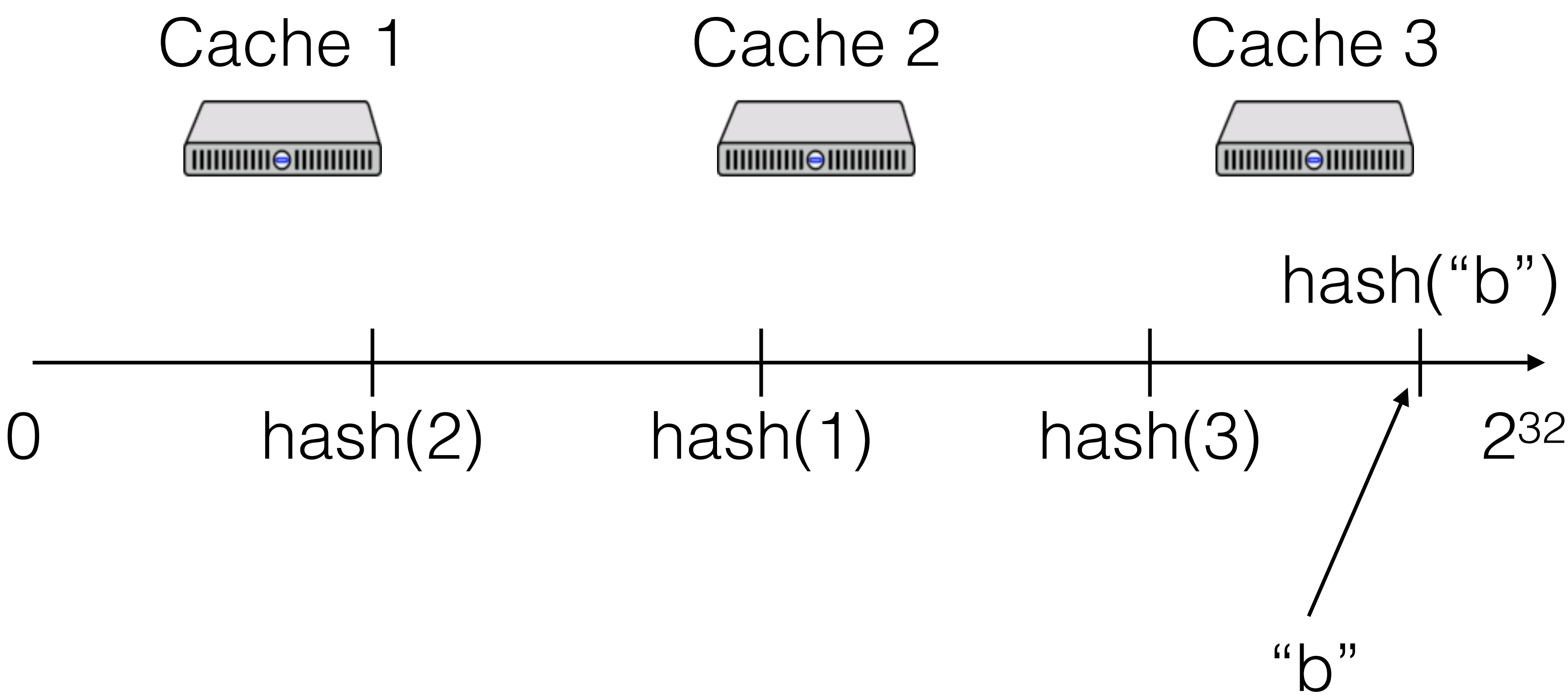


“b”

Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

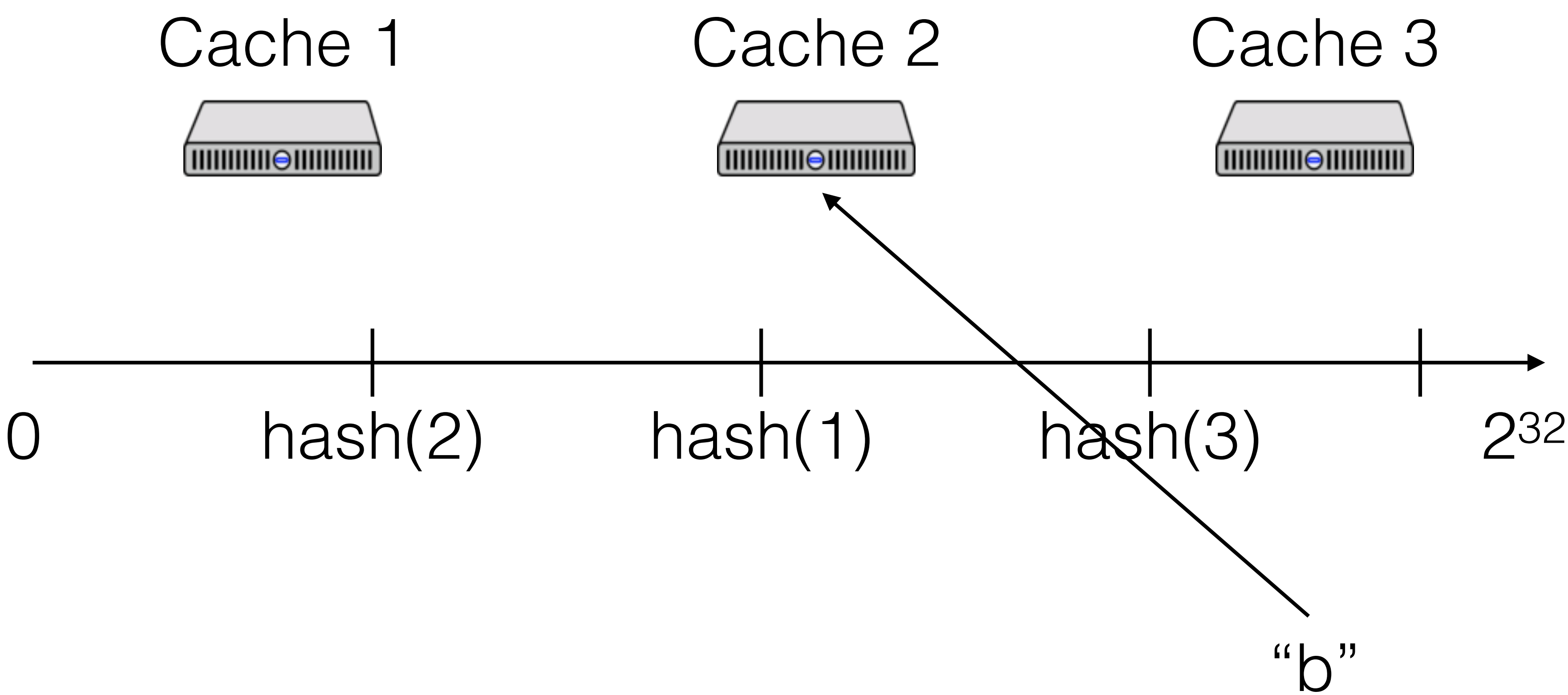
First, hash the node ids



Keys are hashed, go to the "next" node

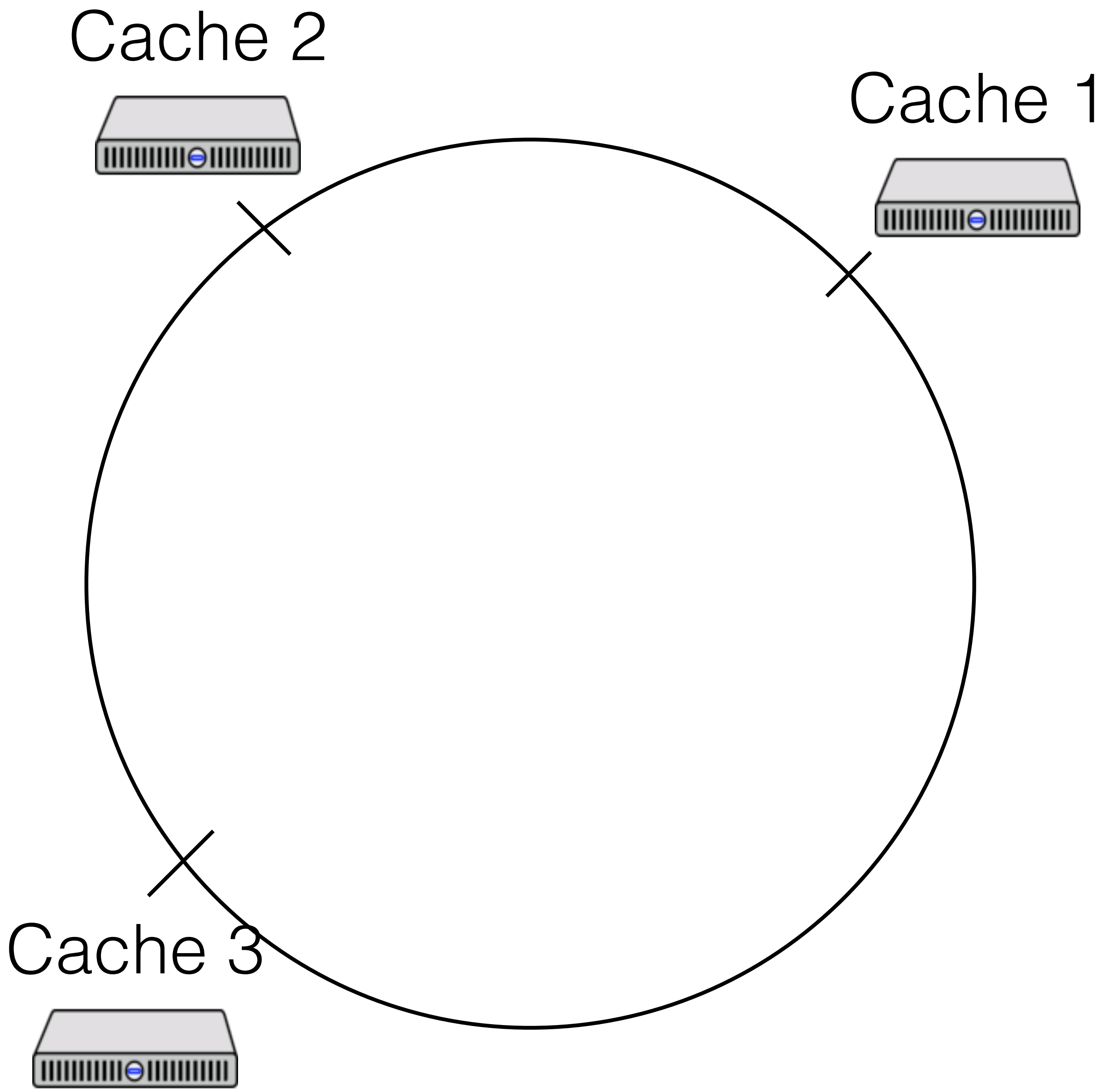
Proposal 3: Consistent Hashing

First, hash the node ids

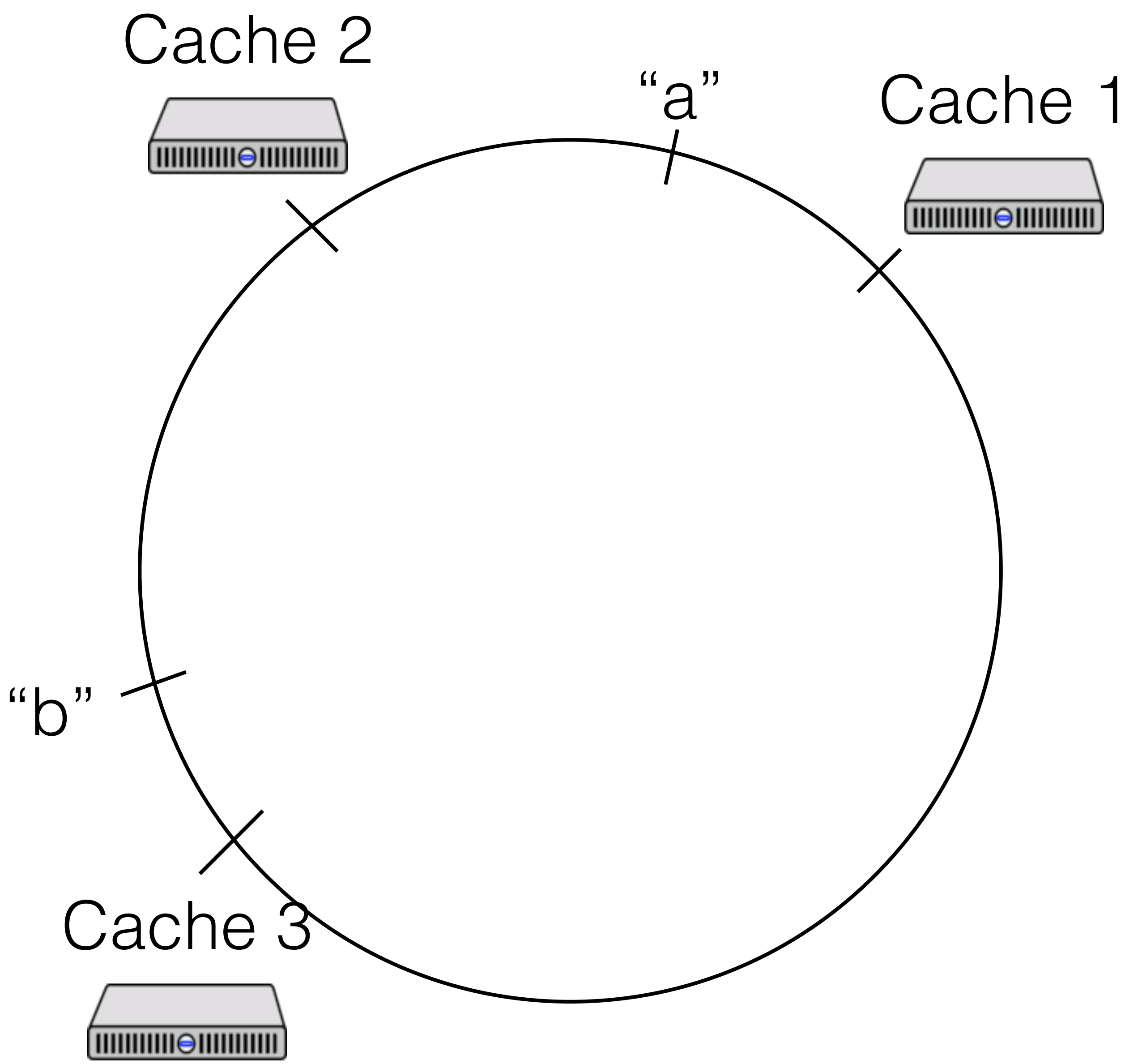


Keys are hashed, go to the "next" node

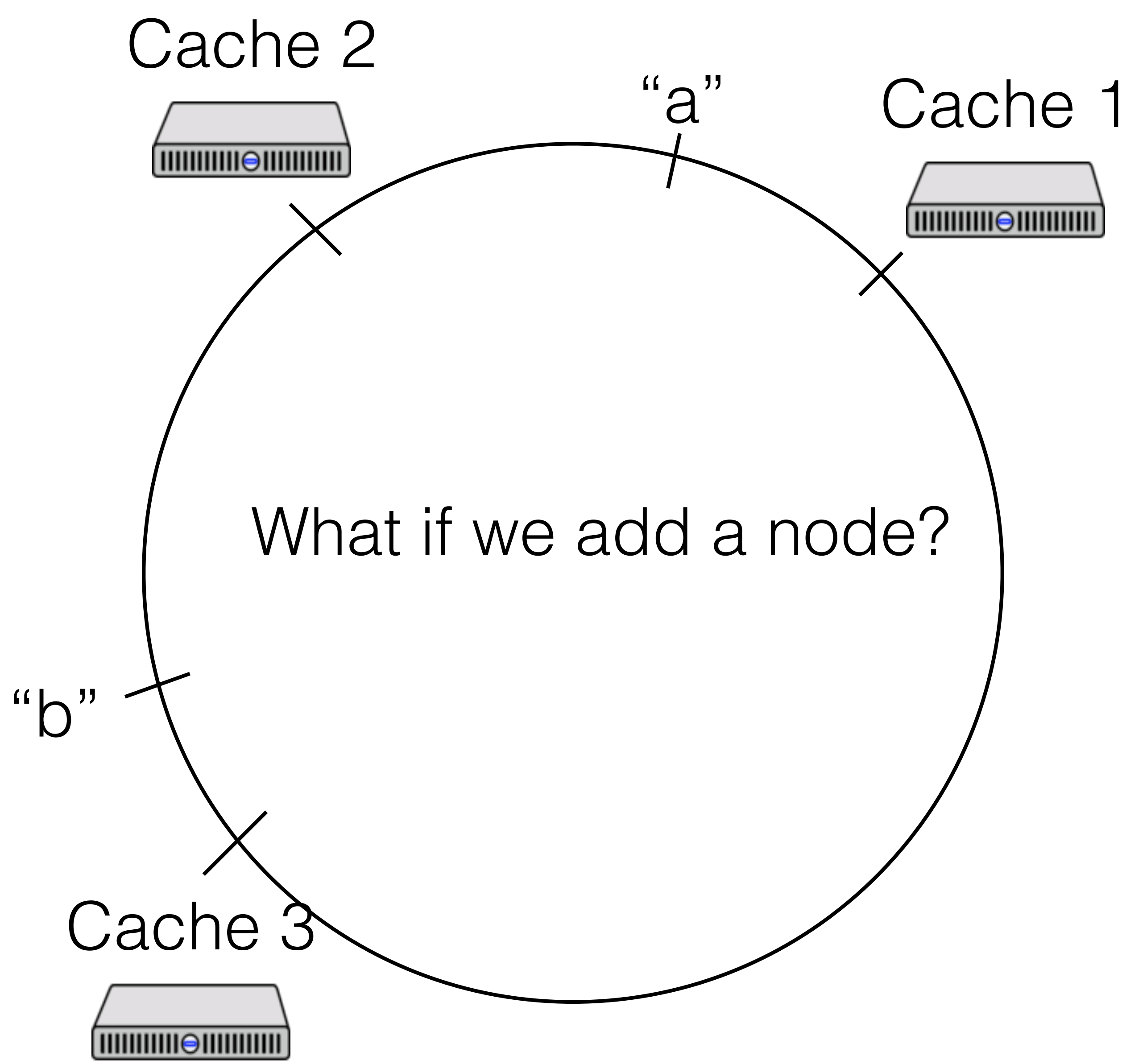
Proposal 3: Consistent Hashing



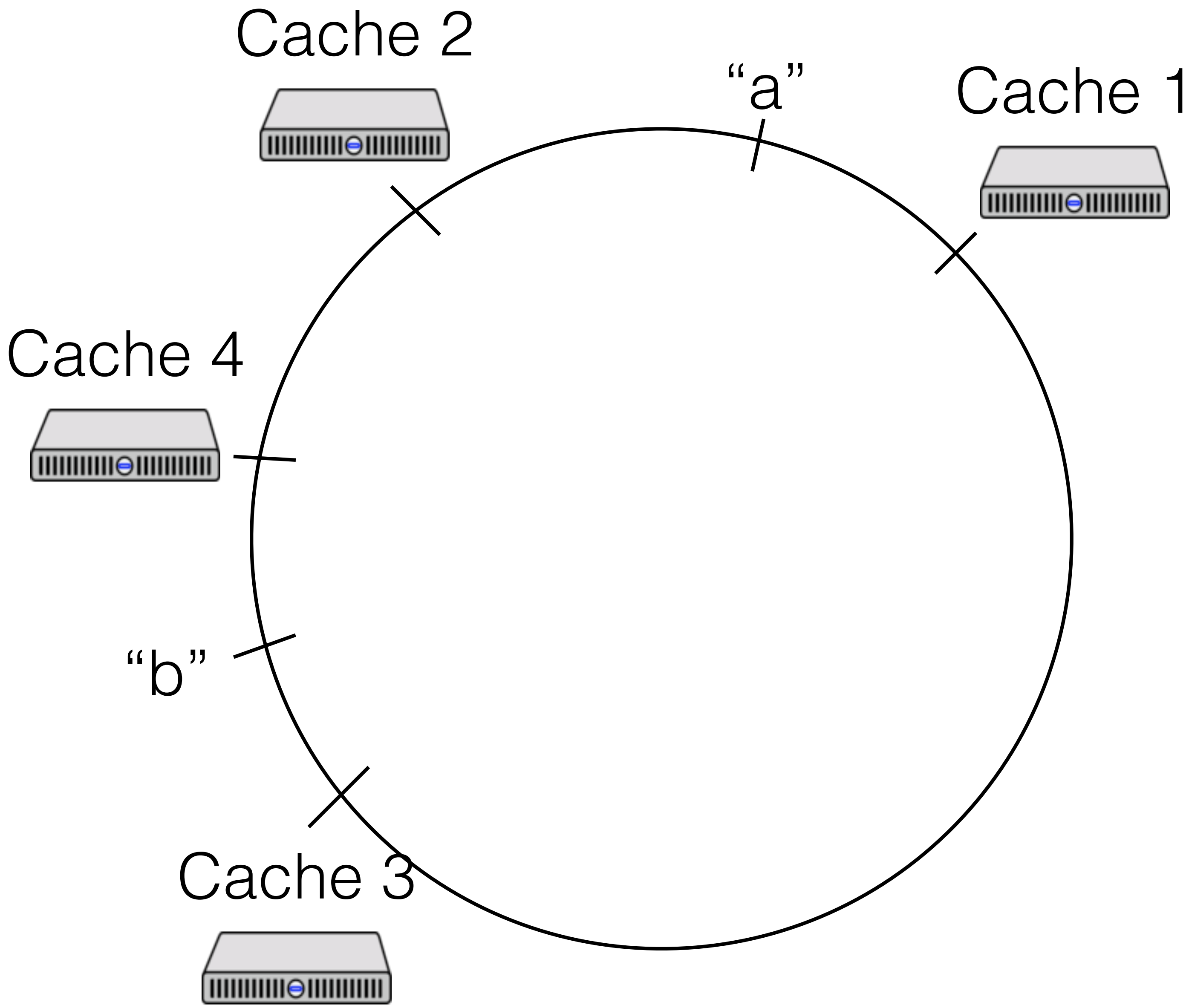
Proposal 3: Consistent Hashing



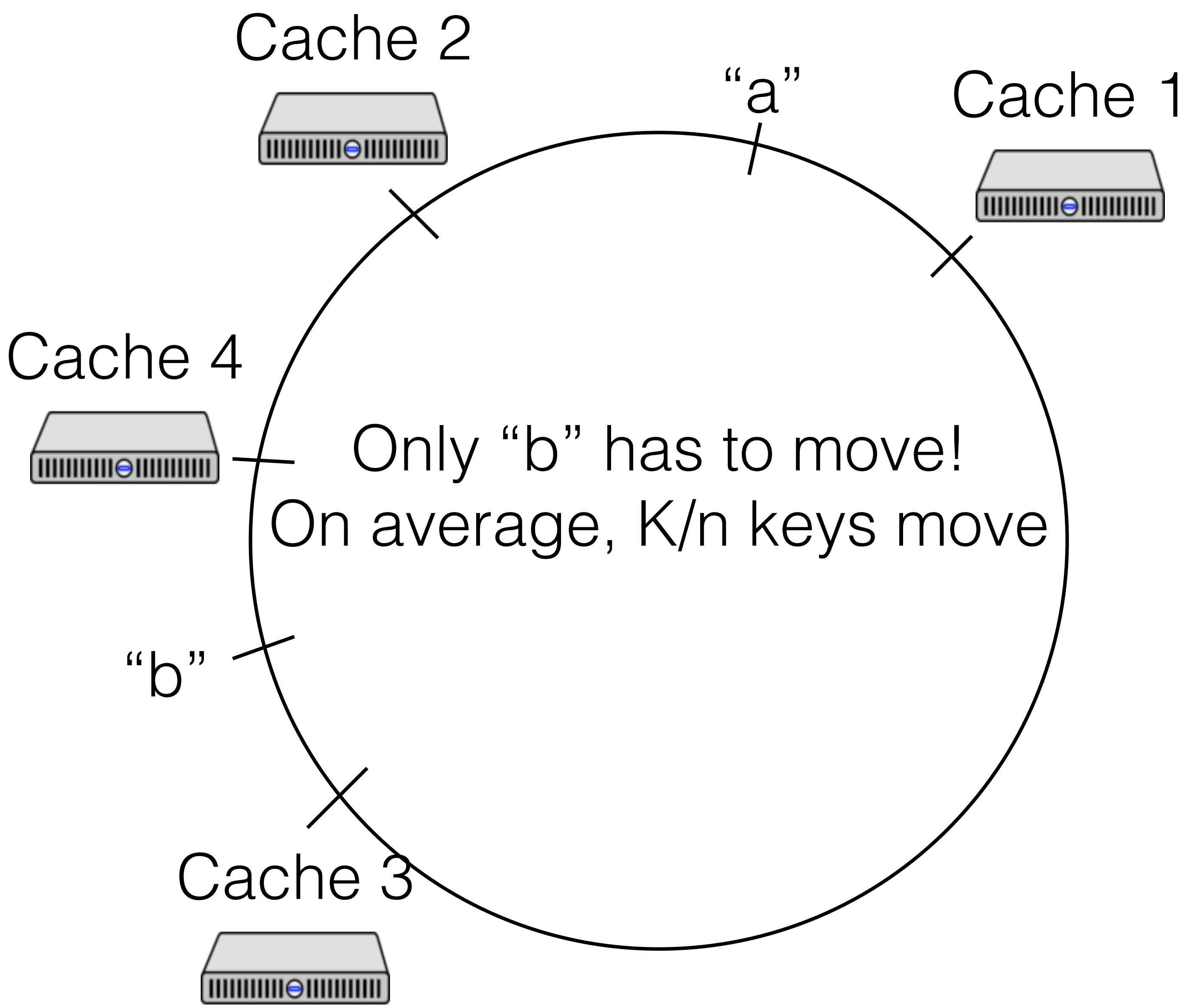
Proposal 3: Consistent Hashing



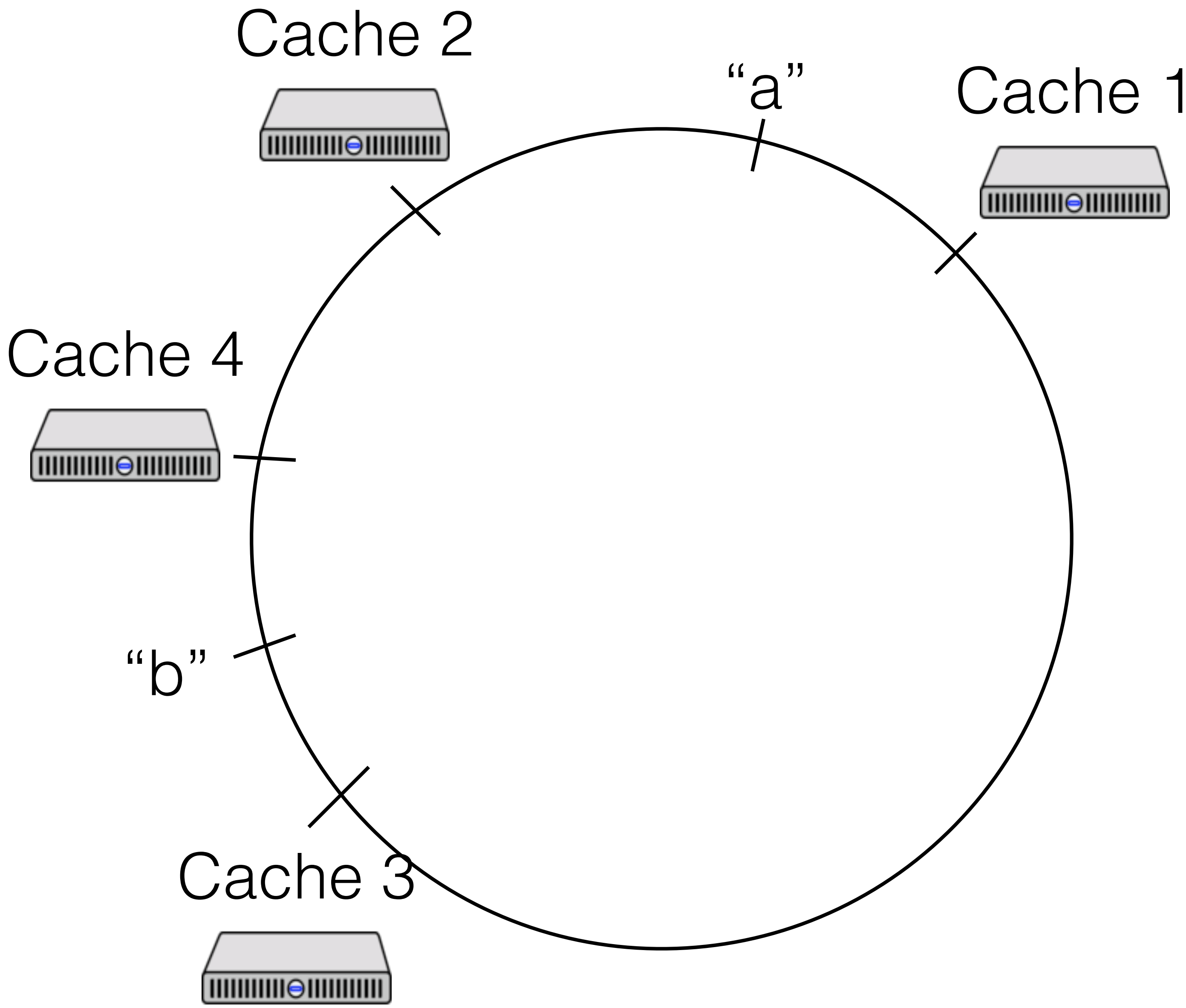
Proposal 3: Consistent Hashing



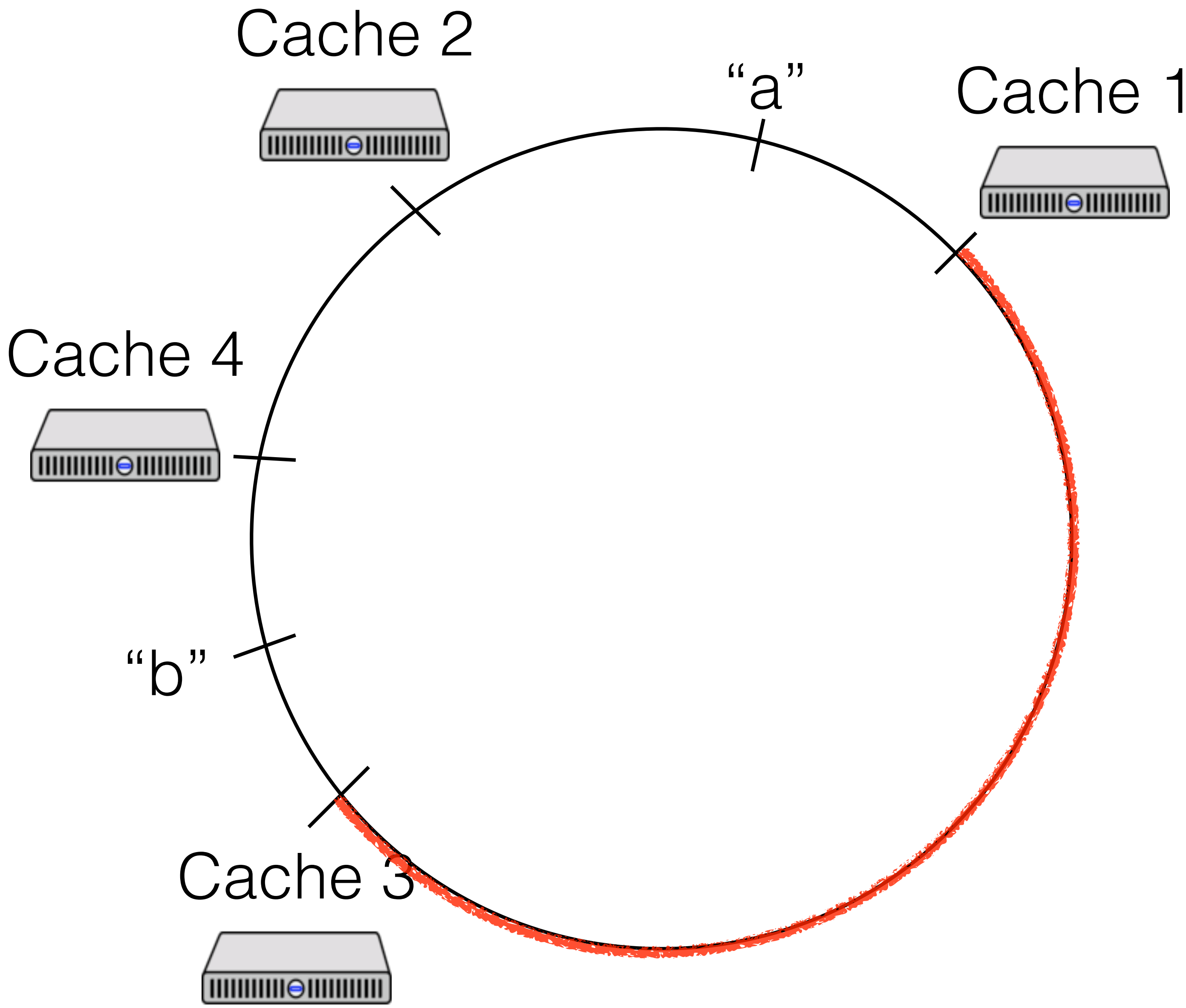
Proposal 3: Consistent Hashing



Proposal 3: Consistent Hashing



Proposal 3: Consistent Hashing



Load Balance

Assume # keys >> # of servers

- For example, 100K users -> 100 servers

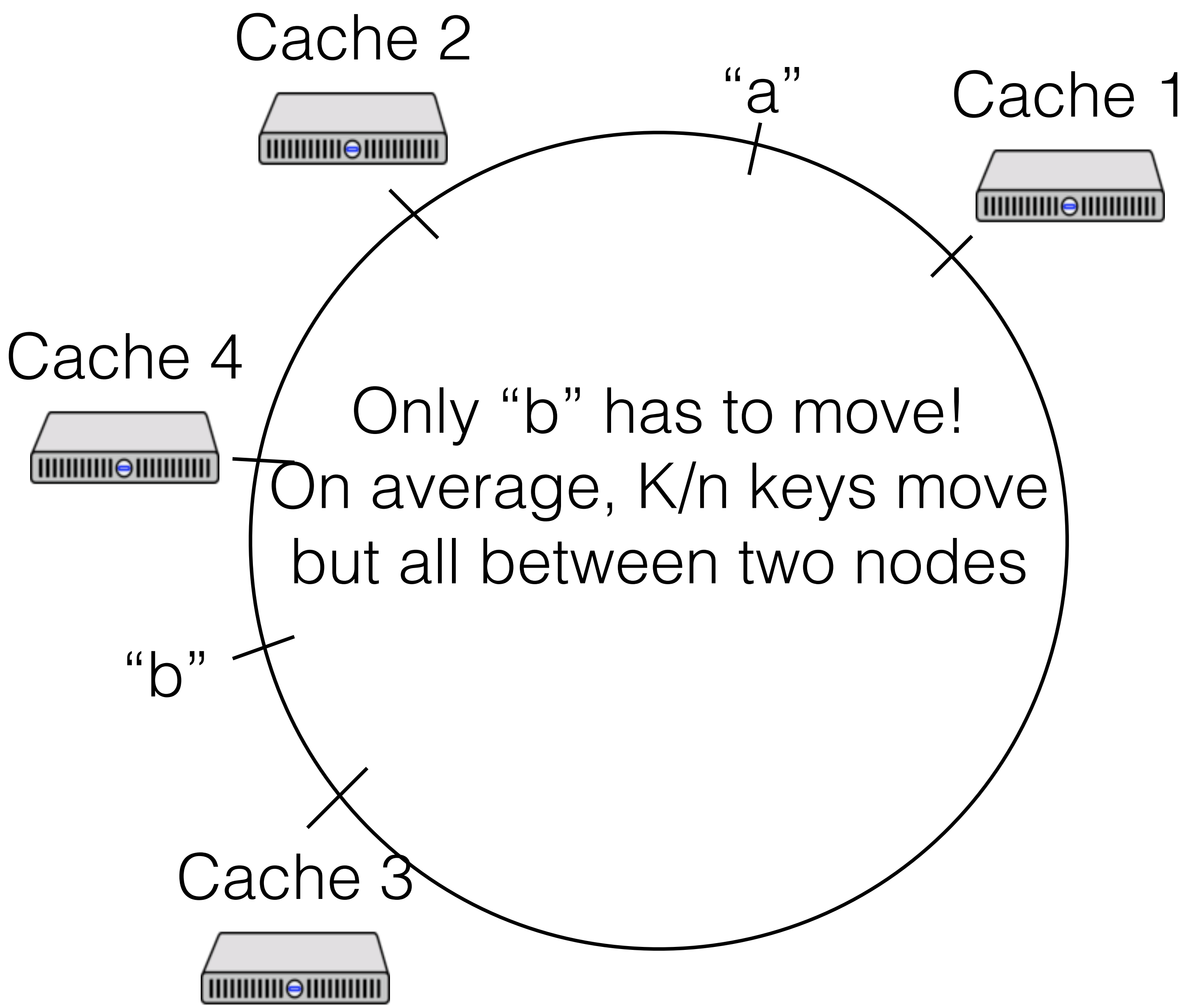
How far off of equal balance is hashing?

- What is typical worst case server?

How far off of equal balance is consistent hashing?

- What is typical worst case server?

Proposal 3: Consistent Hashing



Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

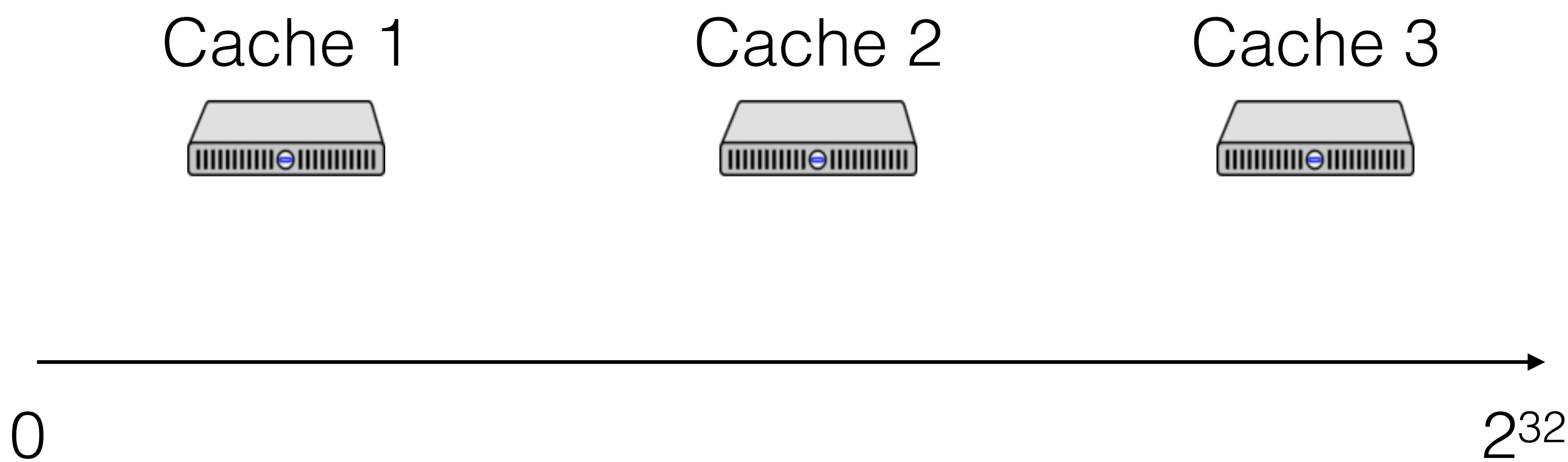
Requirement 3: add/remove node moves only a few keys

Requirement 4: minimize worst case overload

Requirement 5: parcel out work of redistributing keys

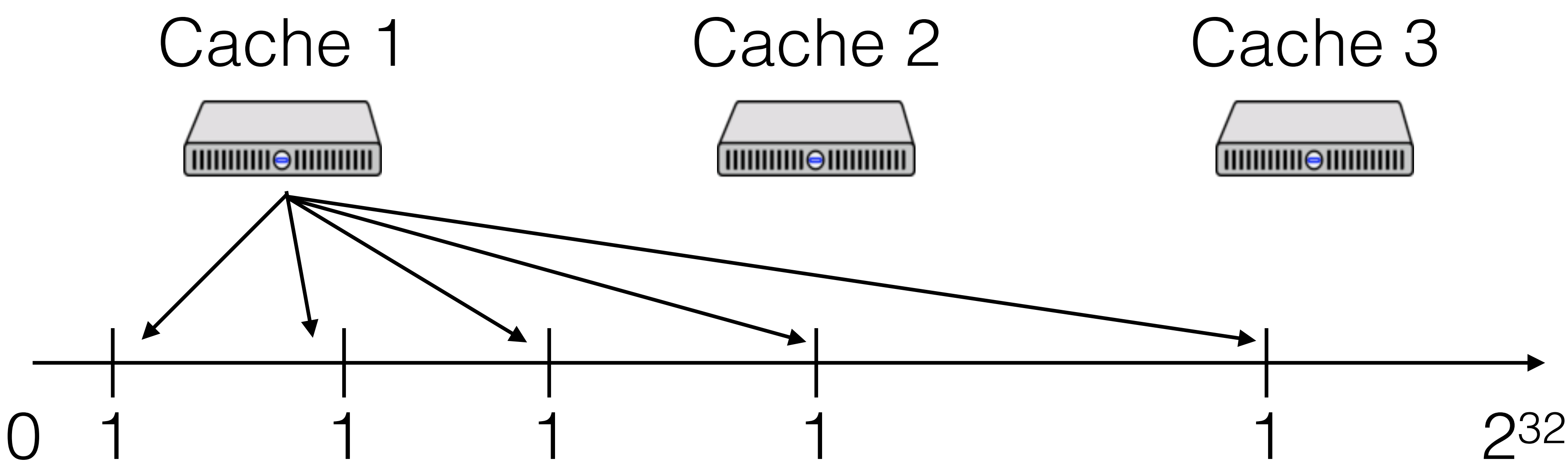
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



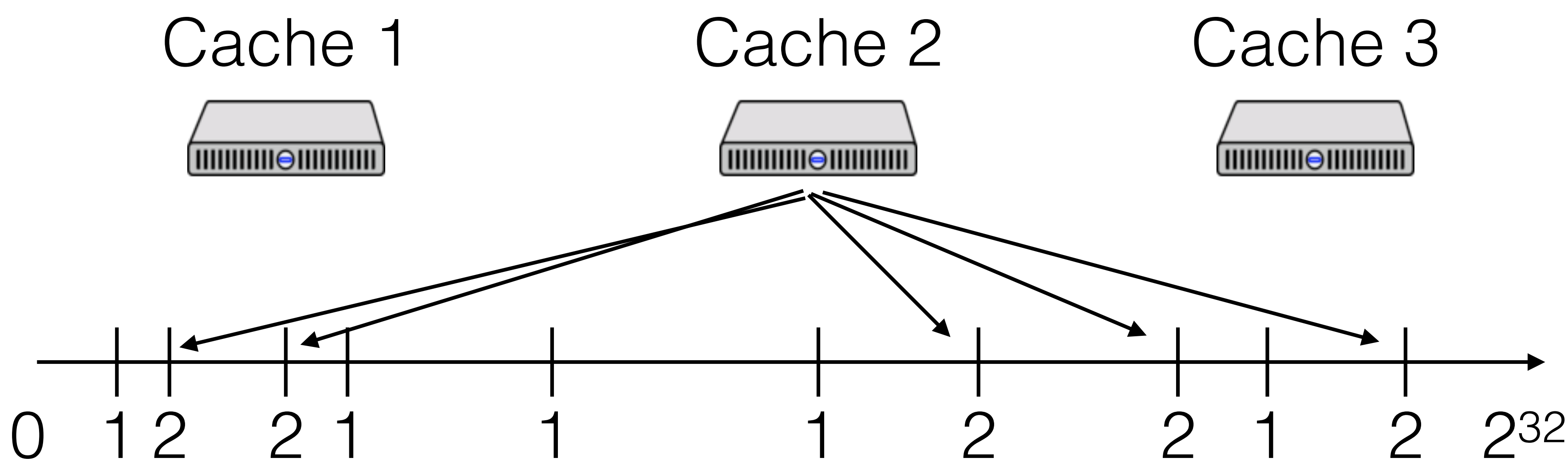
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



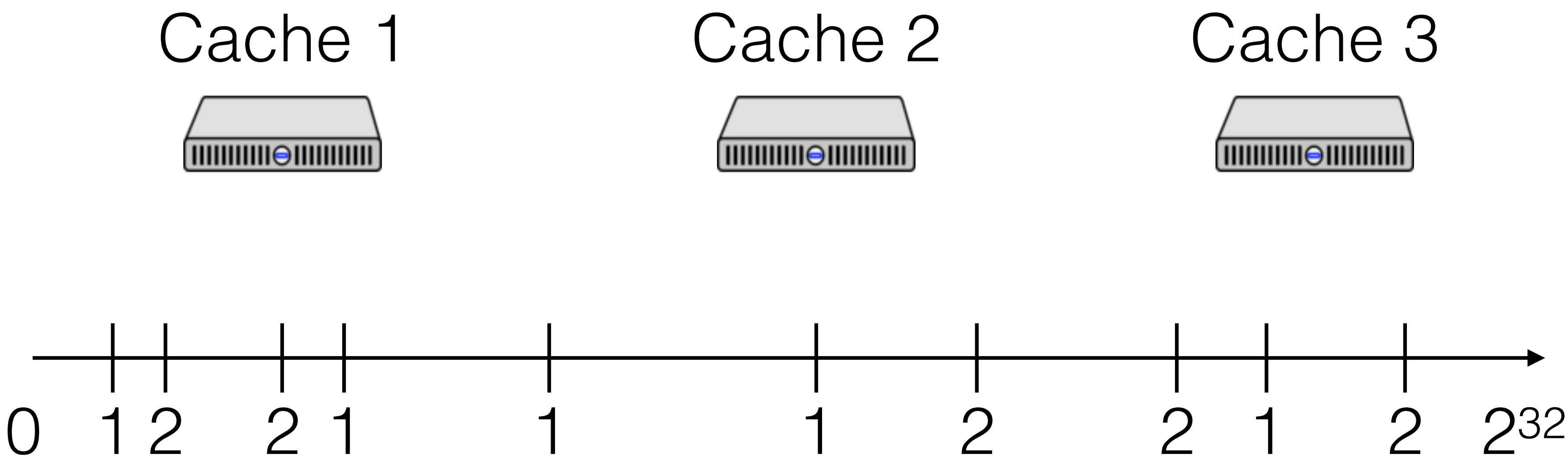
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



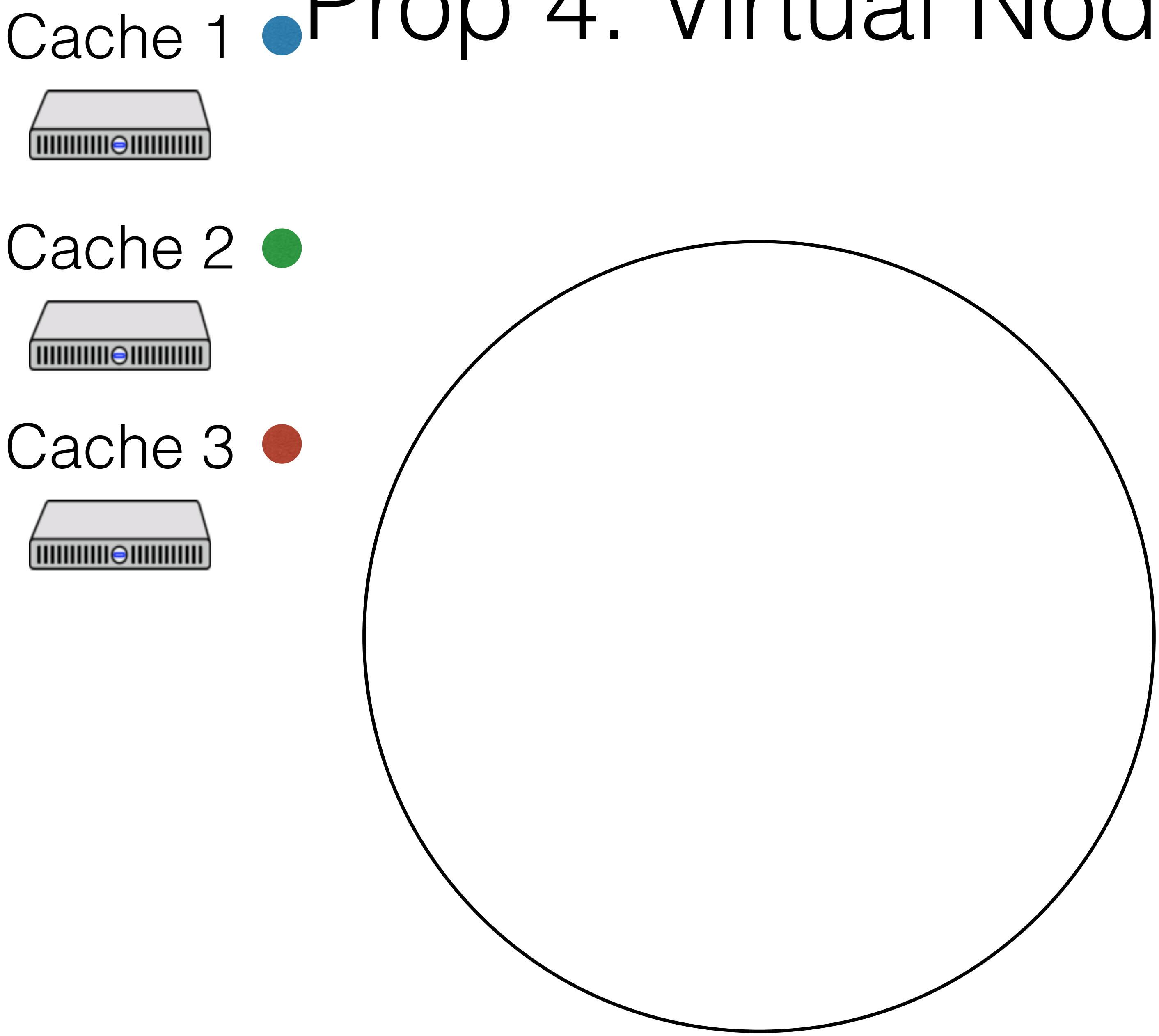
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



As it turns out, hash functions come in families s.t. their members are independent. So this is easy!

Prop 4: Virtual Nodes

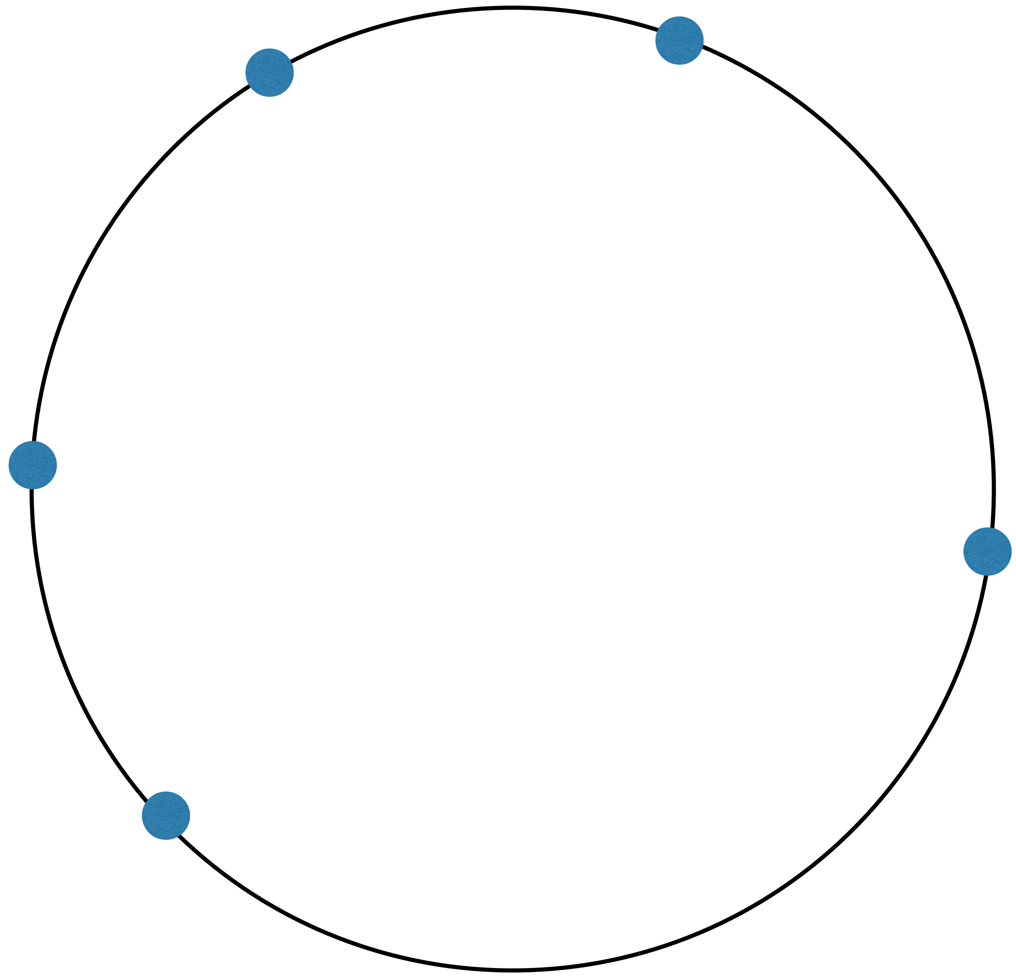


Prop 4: Virtual Nodes

Cache 1

Cache 2

Cache 3

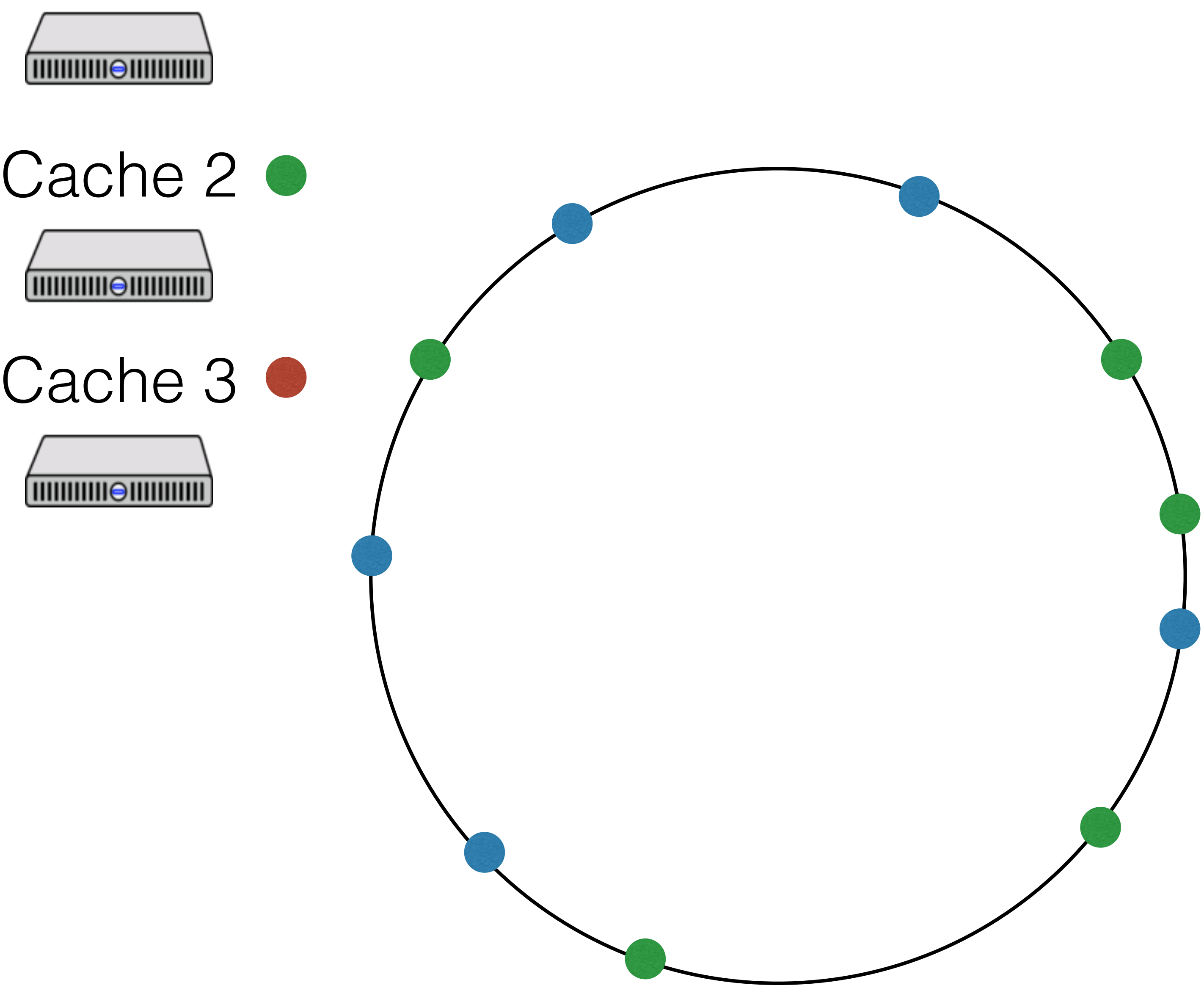


Prop 4: Virtual Nodes

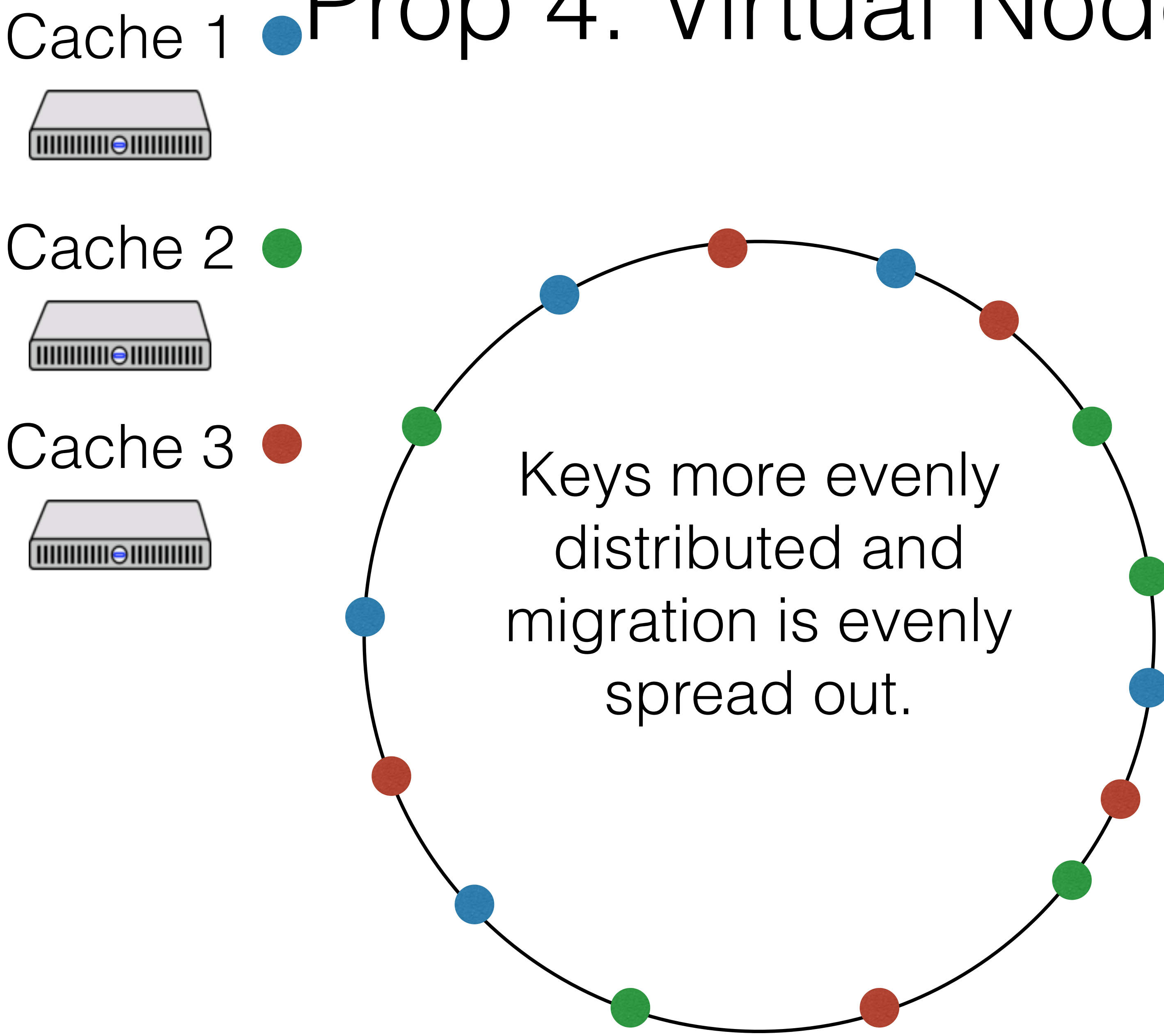
Cache 1

Cache 2

Cache 3



Prop 4: Virtual Nodes



How Many Virtual Nodes?

How many virtual nodes do we need per server?

- to spread worst case load
- to distribute migrating keys

Assume 100000 clients, 100 servers

- 10?
- 100?
- 1000?
- 10000?

Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Requirement 3: add/remove node moves only a few keys

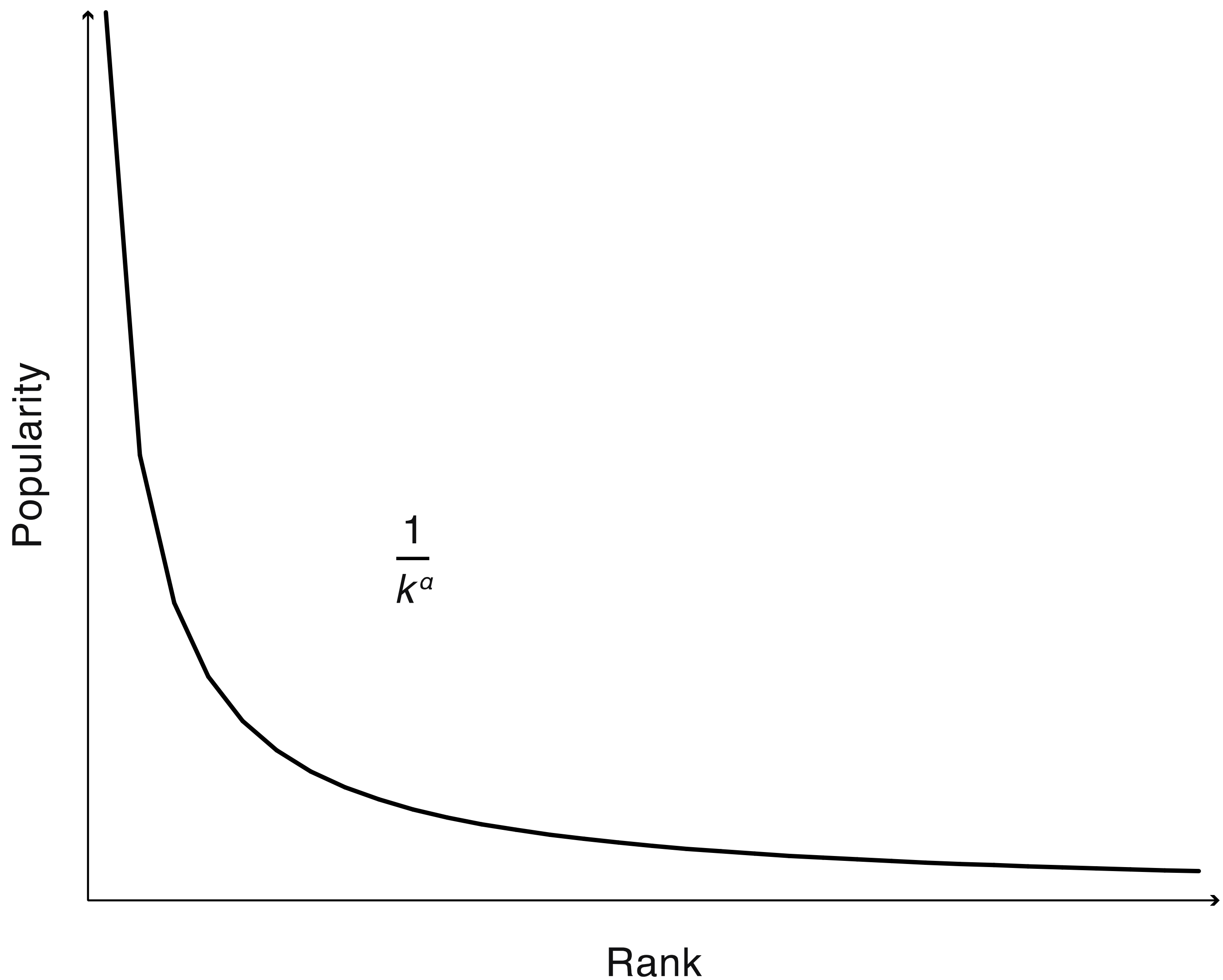
Requirement 4: minimize worst case overload

Requirement 5: parcel out work of redistributing keys

Key Popularity

- What if some keys are more popular than others
- Hashing is no longer load balanced!
- One model for popularity is the Zipf distribution
- Popularity of kth most popular item, $1 < c < 2$
  - $1/k^c$
- Ex: 1, 1/2, 1/3, … 1/100 … 1/1000 … 1/10000

Zipf “Heavy Tail” Distribution



Zipf Examples

- Web pages
- Movies
- Library books
- Words in text
- Salaries
- City population
- Twitter followers
- …

Whenever popularity is self-reinforcing

Popularity changes dynamically: what is popular right now?

Proposal 5: Table Indirection

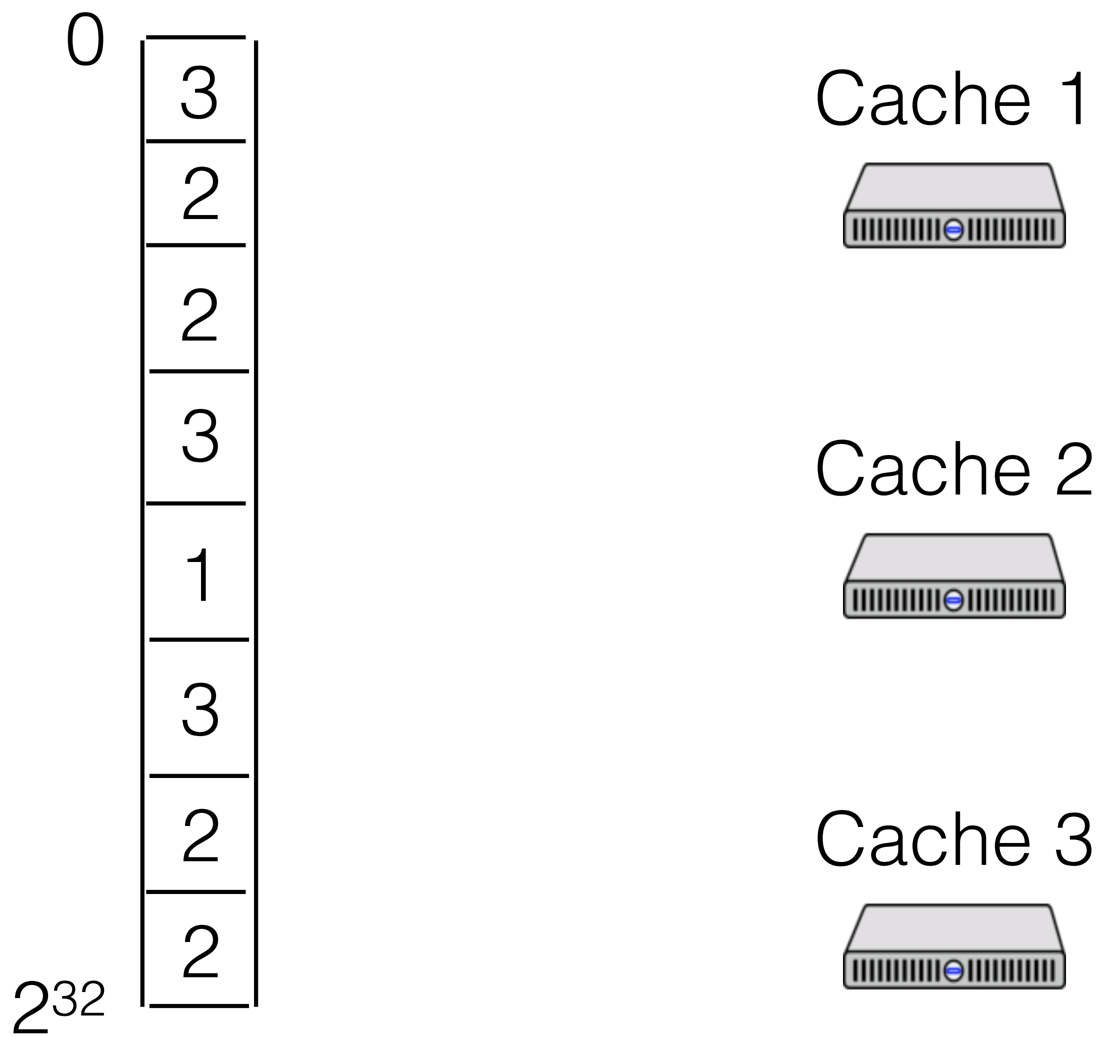
Consistent hashing is (mostly) stateless

- Map is hash function of # servers, # virtual nodes
- Unbalanced with zipf workloads, dynamic load

Instead, put a small table on each client: O(# vnodes)

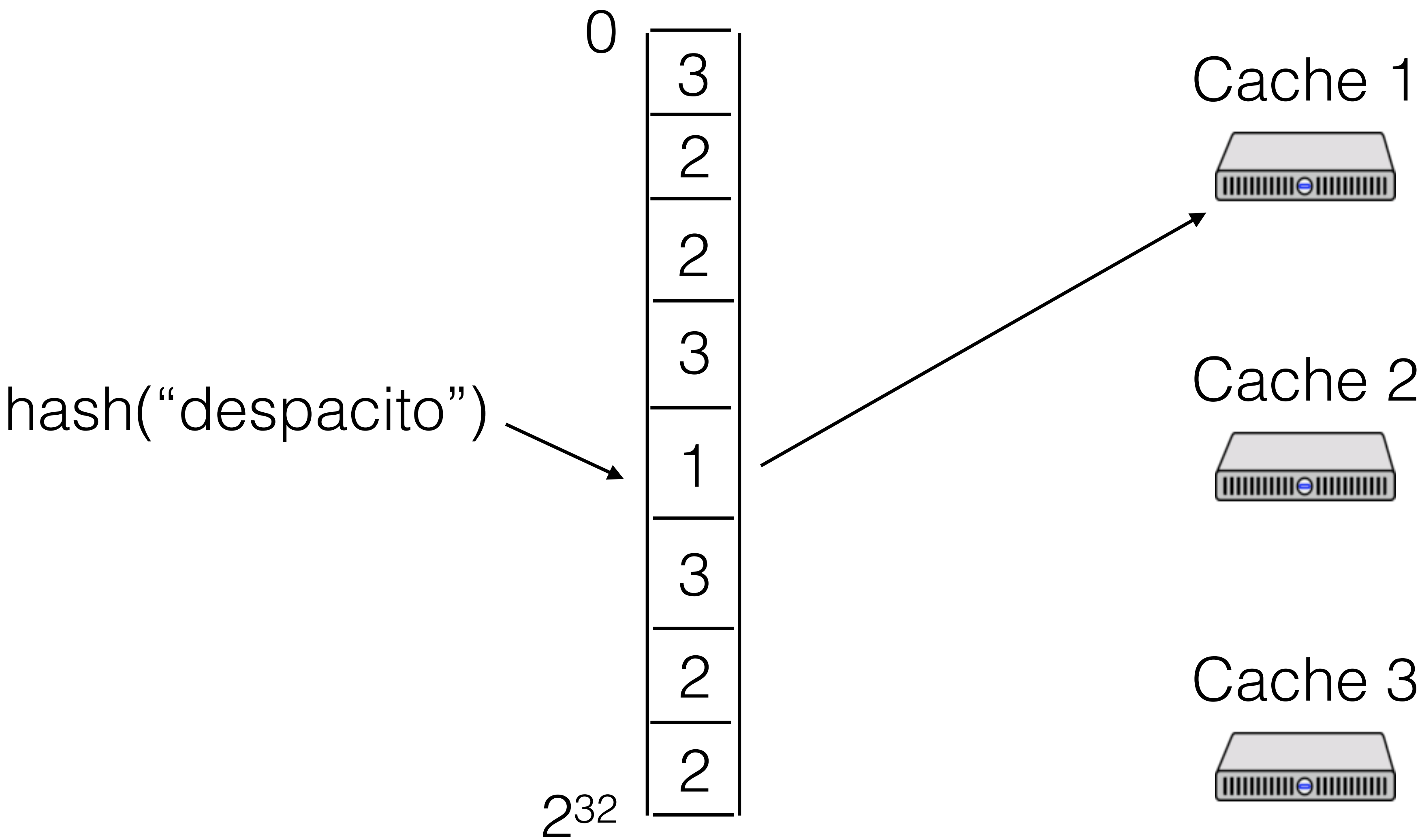
- table[hash(key)] -> server
- Same table on every client
- Shard master adjusts table entries to balance load
- Periodically broadcast new table

Table Indirection



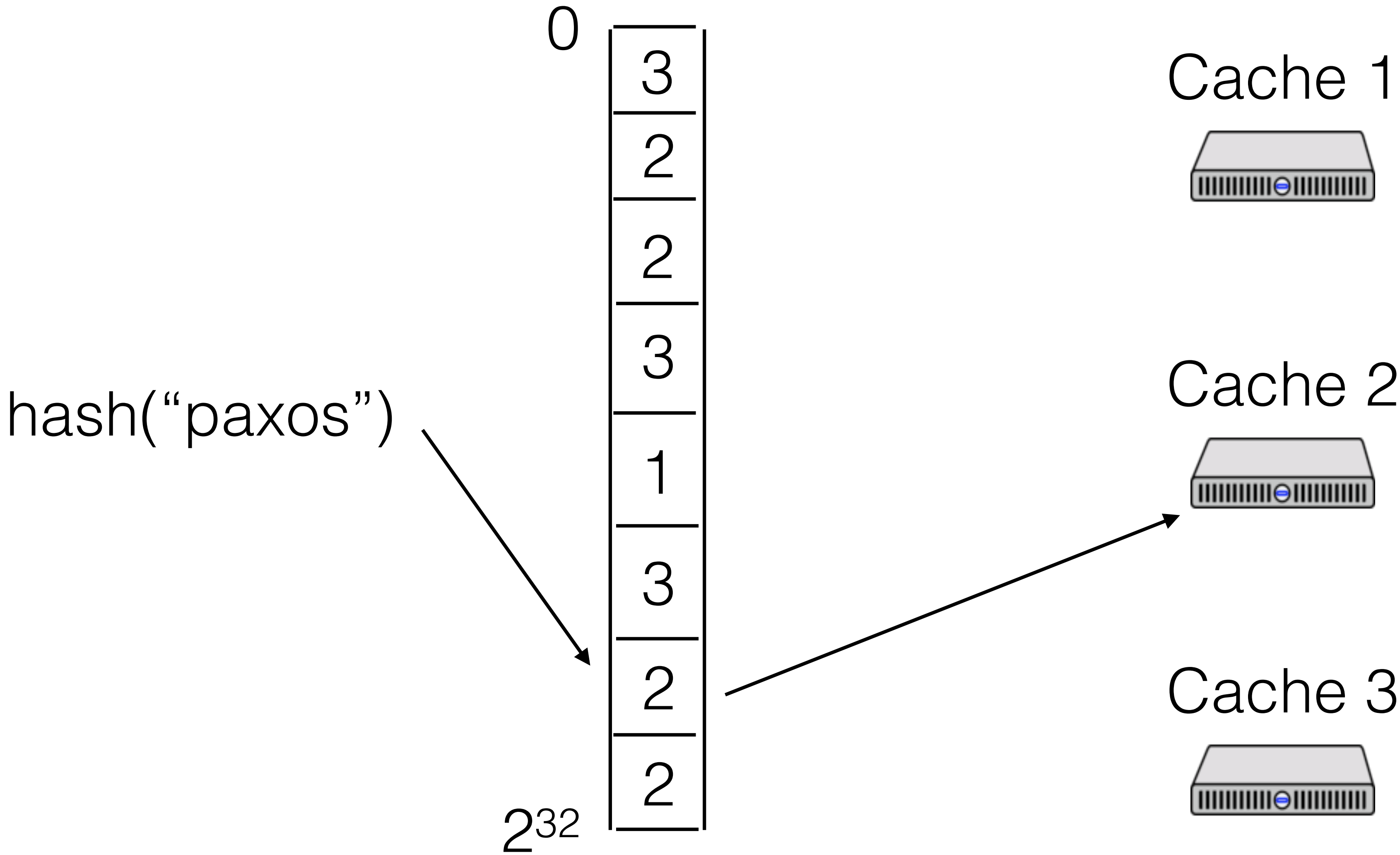
Split hash range into buckets, assign each bucket to a server, busy server gets fewer buckets, can change over time

Table Indirection



Split hash range into buckets, assign each bucket to a server, low load servers get more buckets, can change over time

Table Indirection



Split hash range into buckets, assign each bucket to a server, low load servers get more buckets, can change over time

Proposal 6: Power of Two Choices

Read-only or stateless workloads:

- allow any task to be handled on *one of two* servers
- pair picked at random: hash(k), hash'(k)
- (using consistent hashing with virtual nodes)
- periodically collect data about server load
- send new work to less loaded server *of the two*
- or with likelihood ~ (1 - load)

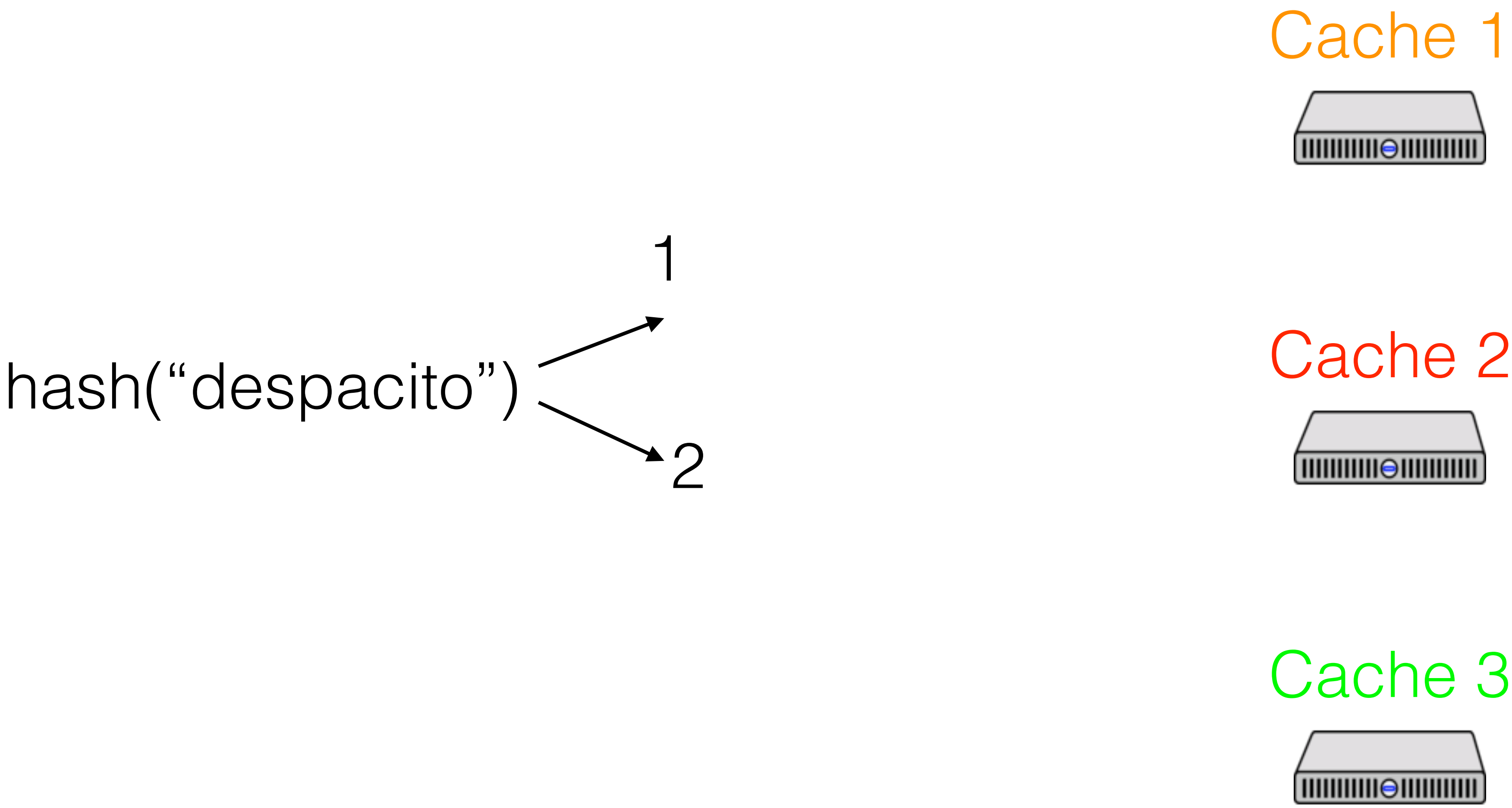
Power of Two Choices

Why does this work?

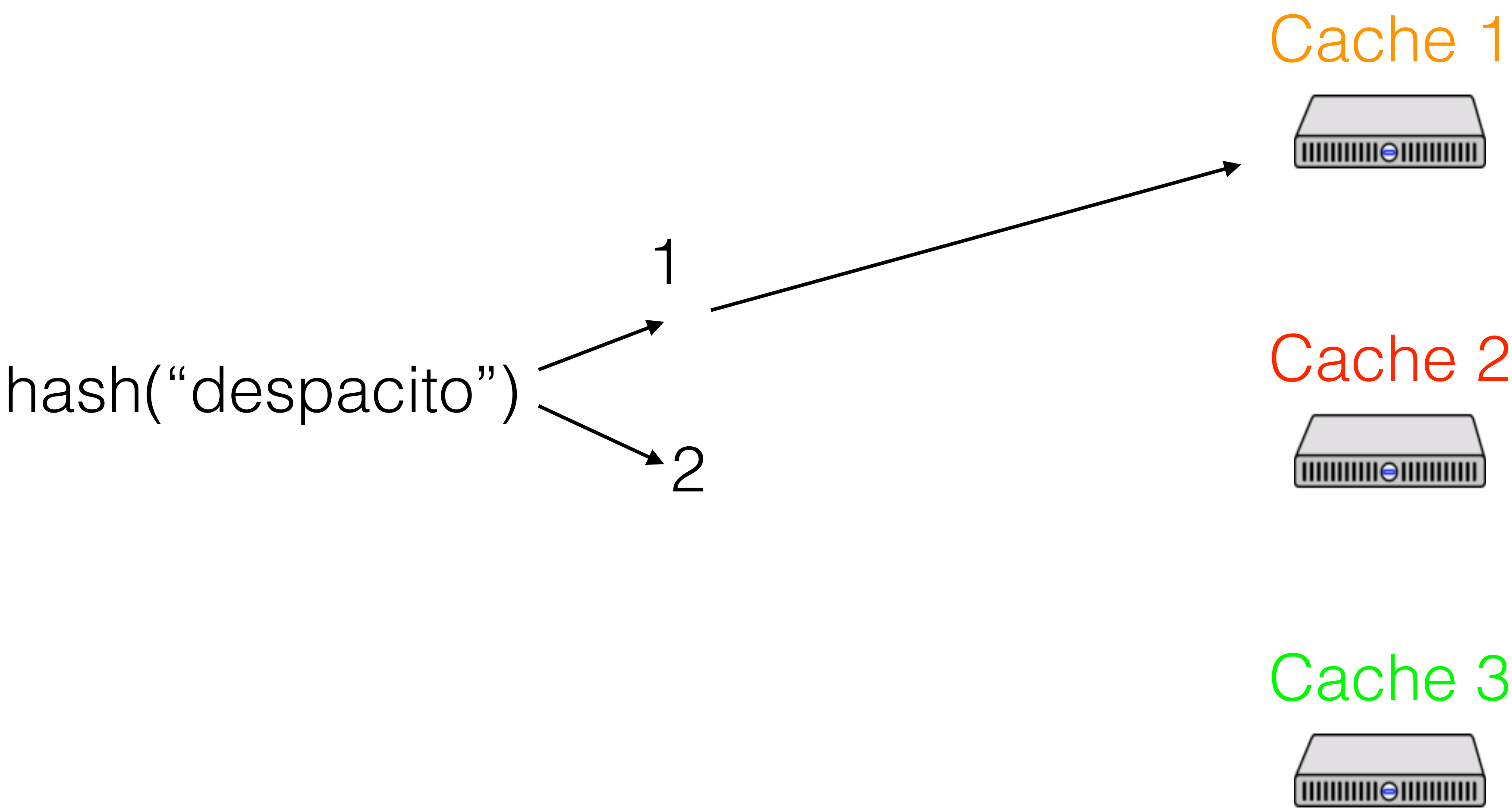
- every key assigned to a different random pair
- suppose k1 happens to map to same server as a popular key k2
- k1’s alternate very likely to be different than k2’s alternate

Generalize: spread very busy keys over more choices

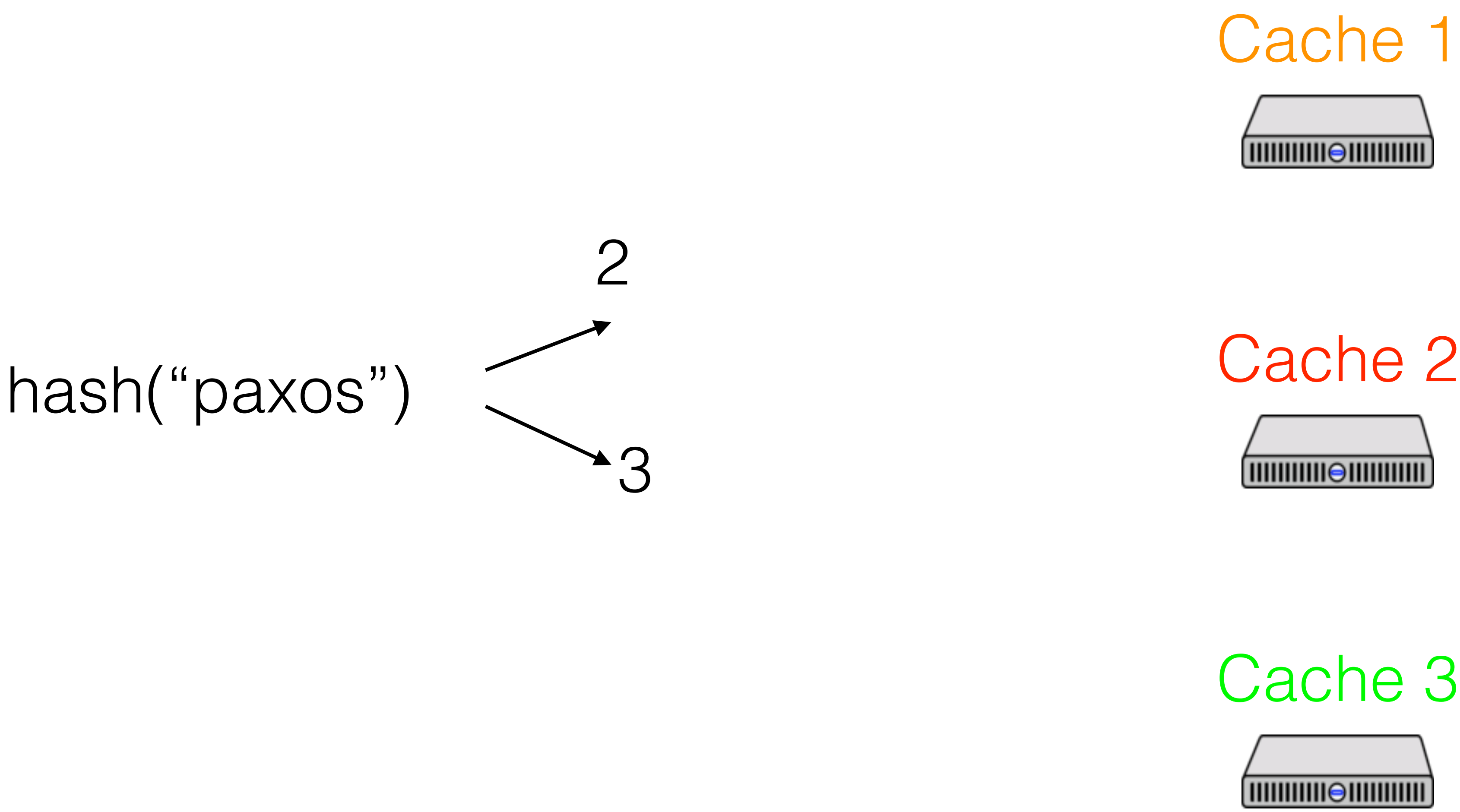
Power of Two Choices



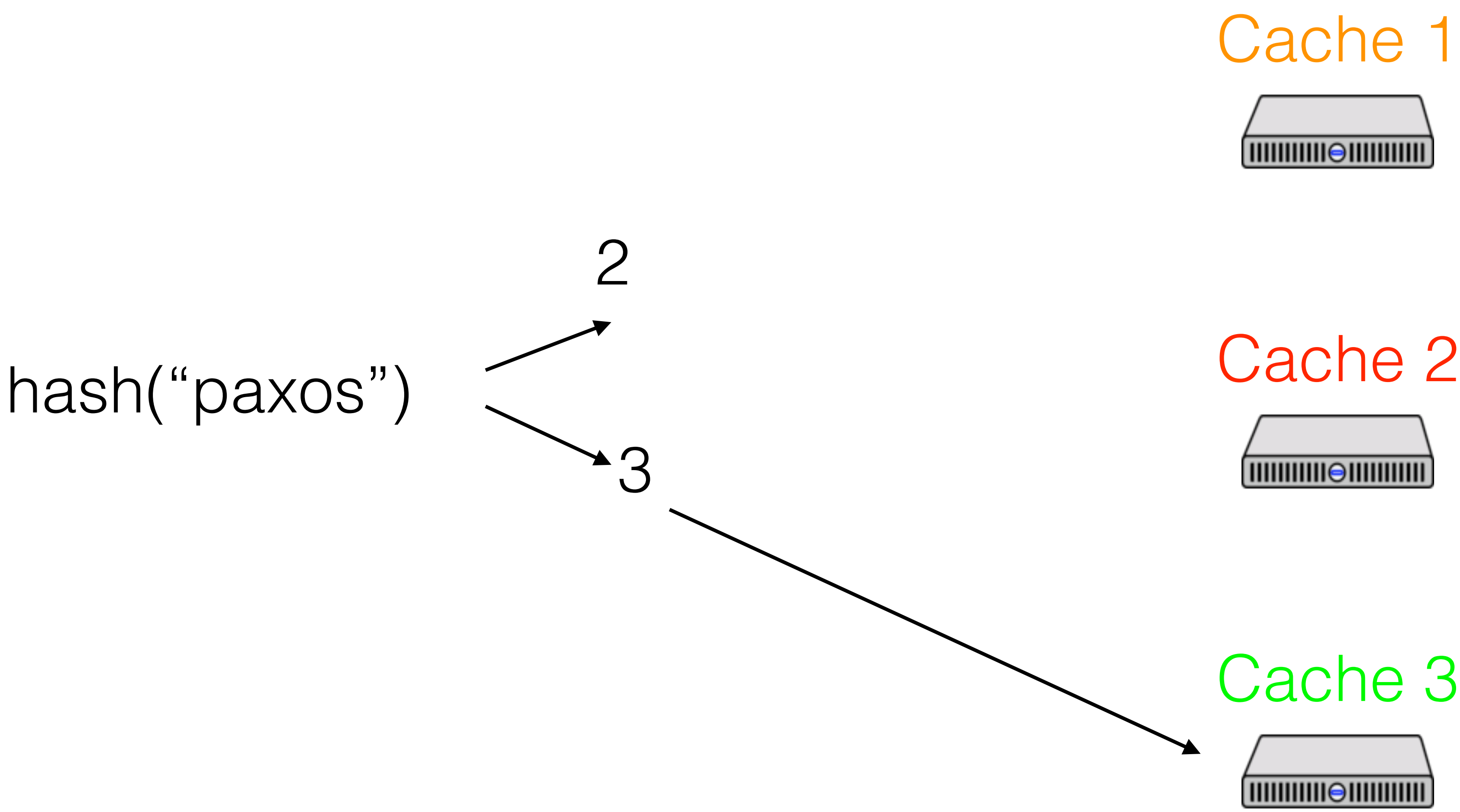
Power of Two Choices



Power of Two Choices



Power of Two Choices



Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Requirement 3: add/remove node moves only a few keys

Requirement 4: minimize worst case overload

Requirement 5: parcel out work of redistributing keys

Requirement 6: balance work even with zipf demand

Next

“Distributed systems in practice”

- Memcache: scalable caching layer between stateless front ends and storage

- GFS: scalable distributed storage for stream files

- BigTable: scalable key-value store

- Spanner: cross-data center transactional key-value store

Thursday

Yegge on Service-Oriented Architectures

- Steve Yegge, prolific programmer and blogger
- Moved from Amazon to Google
- Reading is an accidentally-leaked memo about differences between Amazon’s and Google’s system architectures (at that time)
- SOA: separate applications (e.g. Google Search) into many primitive services, run internally as products