Consistent Hashing

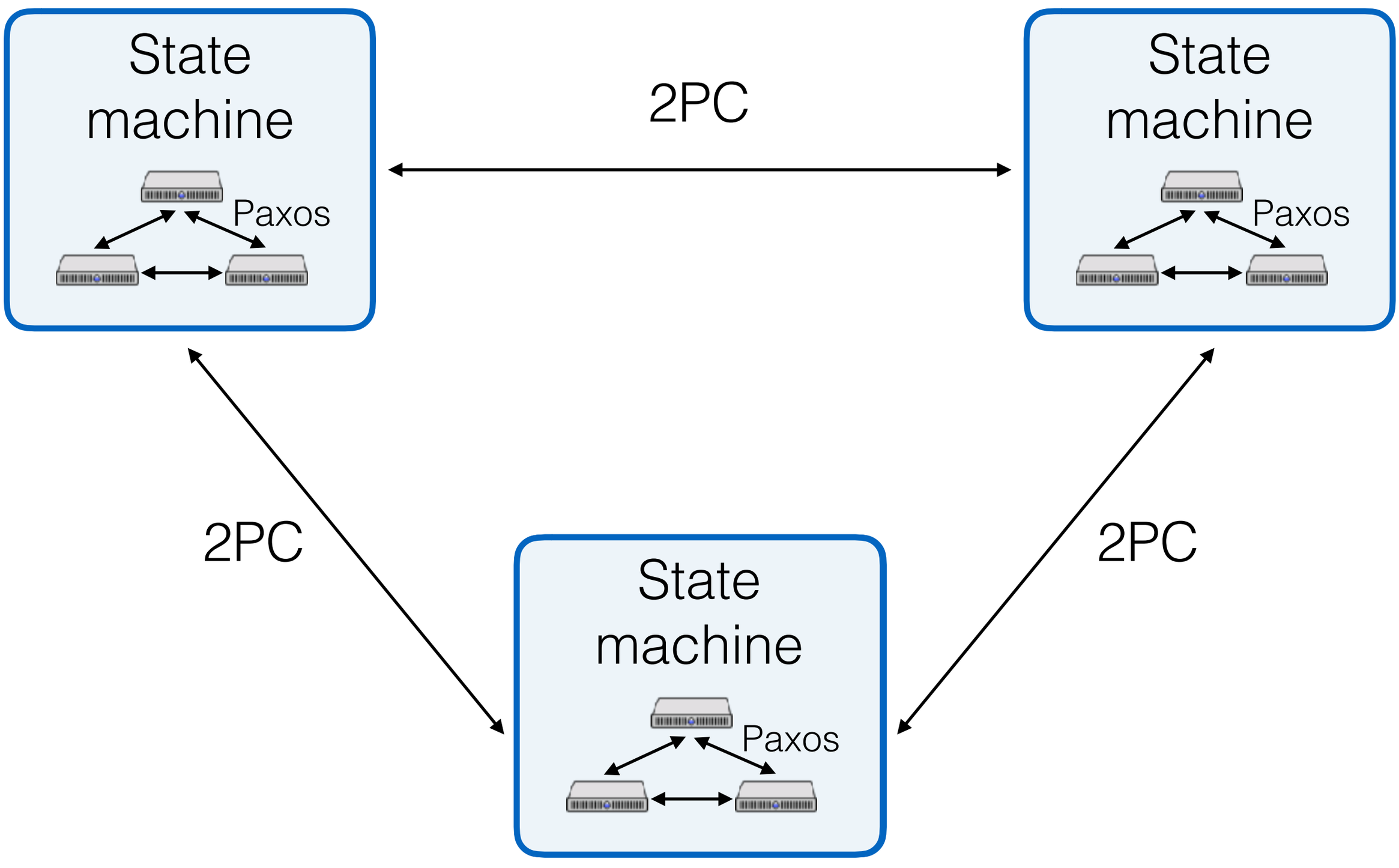
Doug Woos

Logistics notes

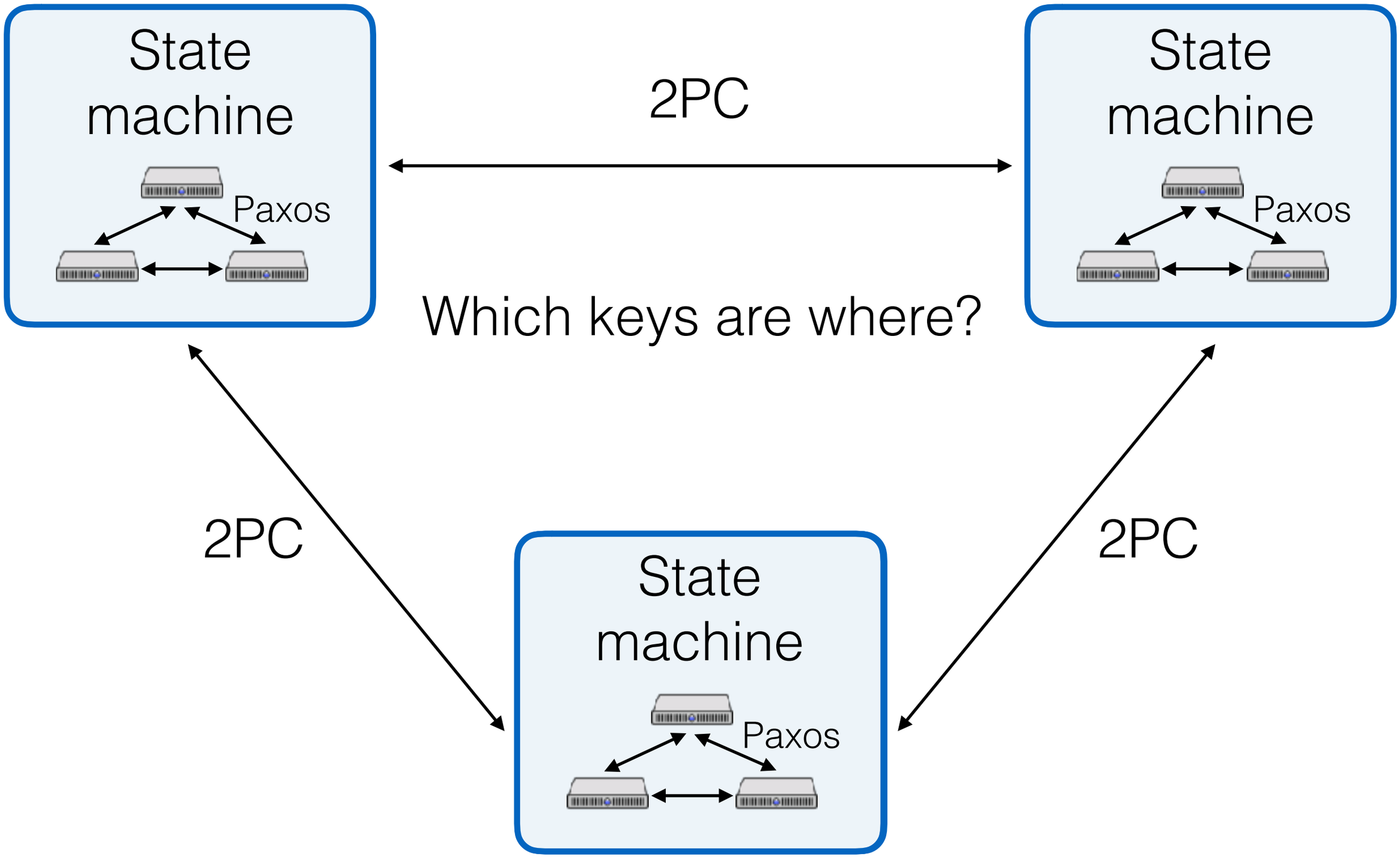
PS3 out later today

Lab 3a due Monday

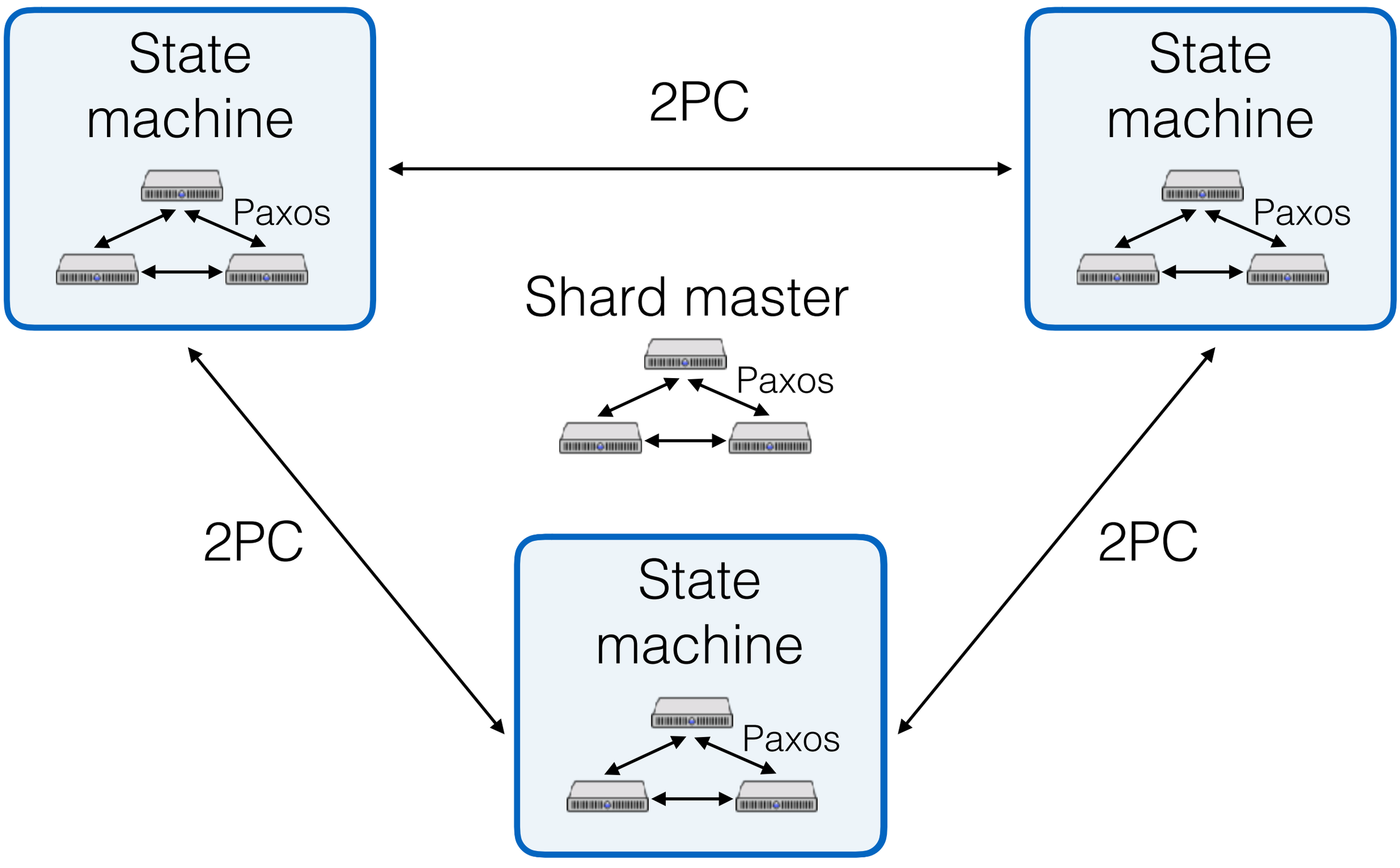
Replicated, Sharded Database



Replicated, Sharded Database



Lab 4 (and others!)



Highly available stores

Want to avoid communication with shard master

Shards operate independently

So: how to ensure clients know who to talk to?

Another scenario

Client

Another scenario

Doug Woos

GET index.html

Client

Another scenario


Doug Woos

Advising

Publications

index.html

Client

Another scenario

Doug Woos

Advising

Publications

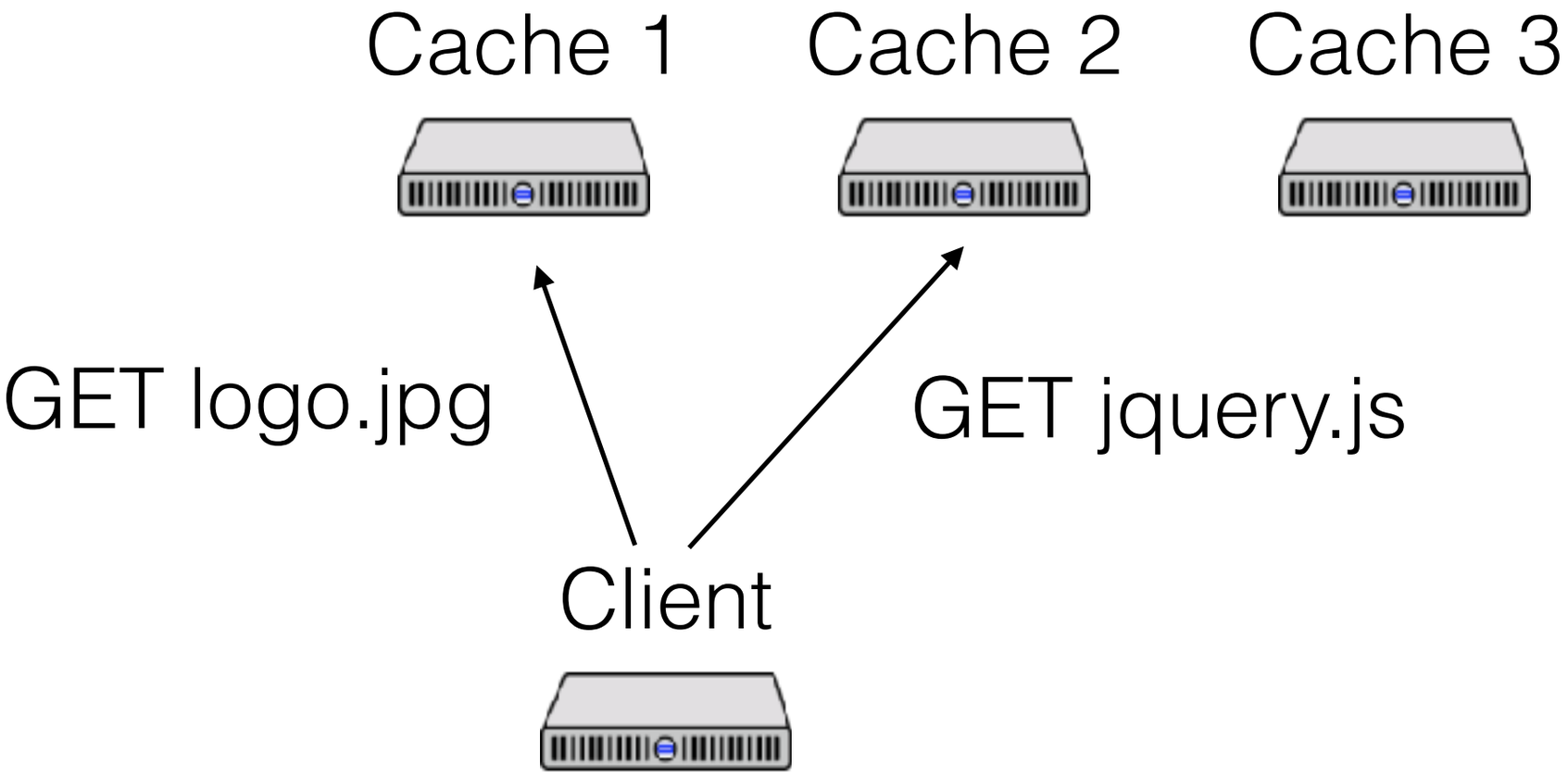
index.html

Links to: logo.jpg, jquery.js, …

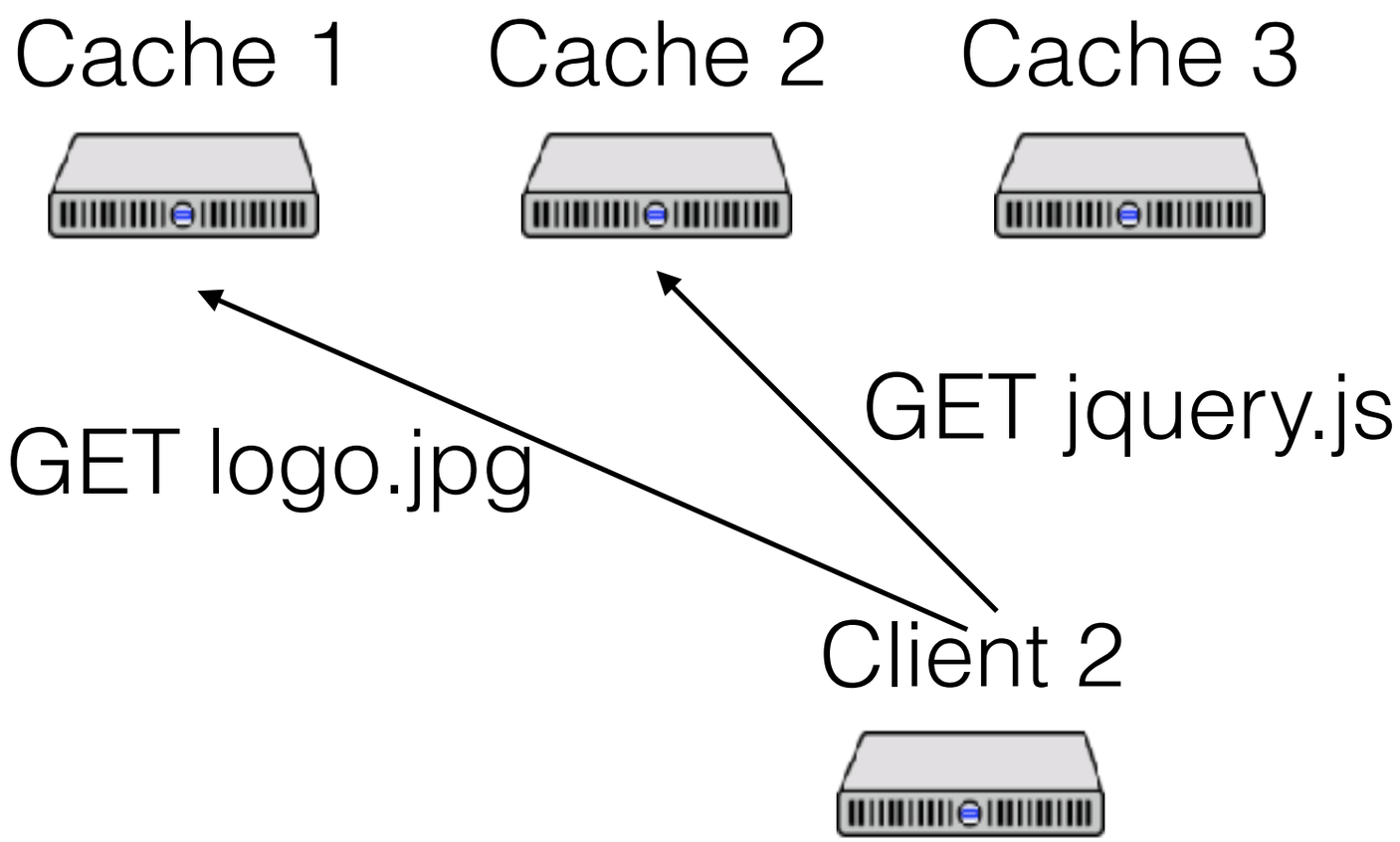
Client

Another scenario

Doug Woos



Another scenario



What's in common?

Want to assign keys to servers w/o communication

Requirement 1: clients all have same assignment

Proposal 1

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

Cache 1 Cache 2 Cache 3

“a”, “d”, “ab” “b” “c”

Proposal 1

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

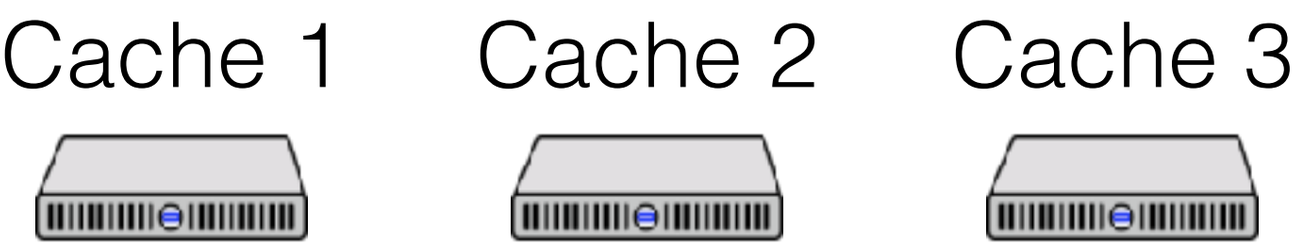
Cache 1 Cache 2 Cache 3

“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?

- Likely to have distribution issues

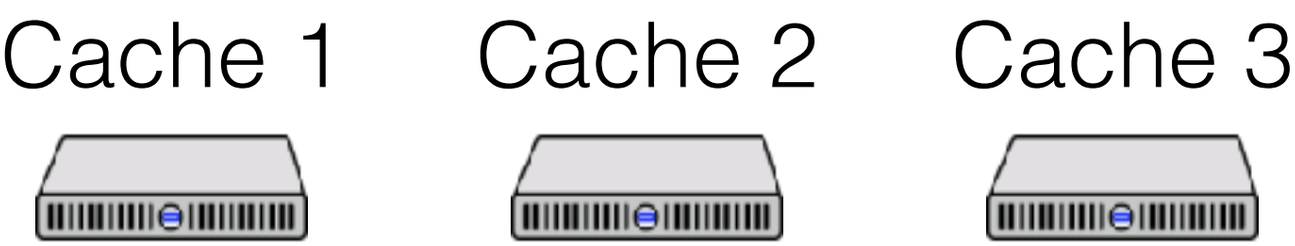
Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Proposal 2

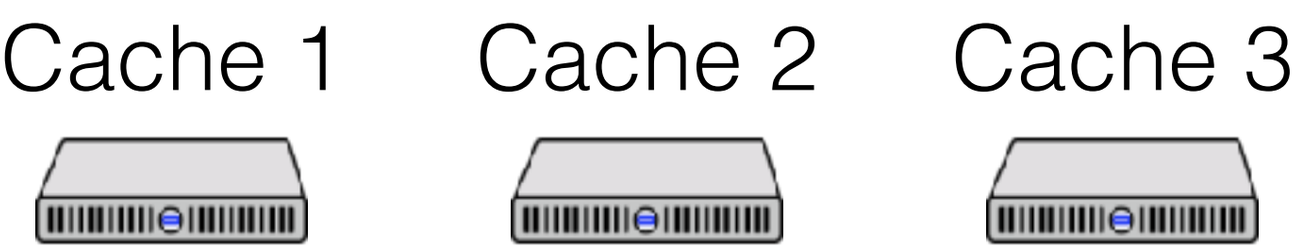
For *n* nodes, a key *k* goes to *hash(k)* mod *n*



Hash distributes keys uniformly

Proposal 2

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

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

Cache 1 Cache 2 Cache 3 Cache 4

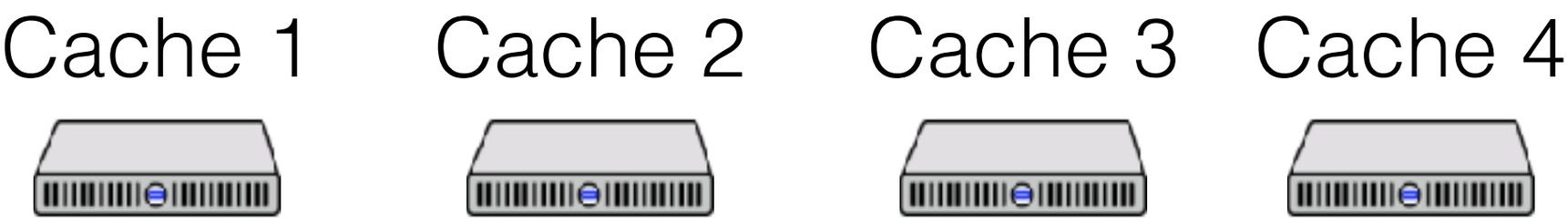
h(“a”)=1 *h*(“abc”)=2 *h*(“b”)=3

Hash distributes keys uniformly

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

Proposal 2

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

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: can add/remove nodes w/o redistributing too many keys

Proposal 3

First, hash the node ids

Proposal 3

First, hash the node ids

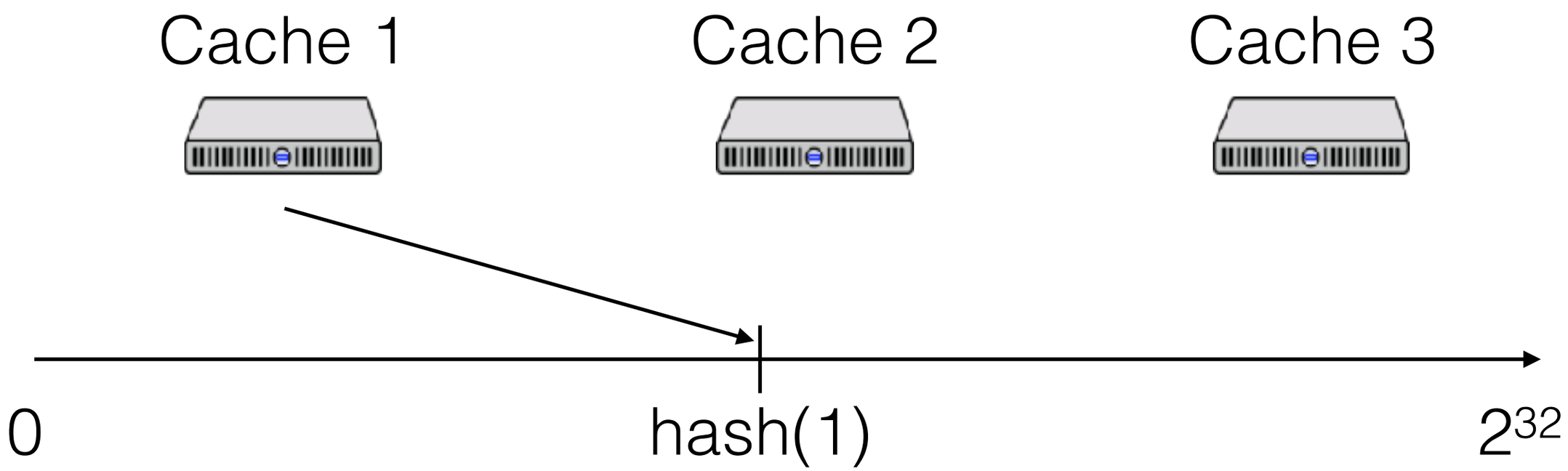
Cache 1

Cache 2

Cache 3

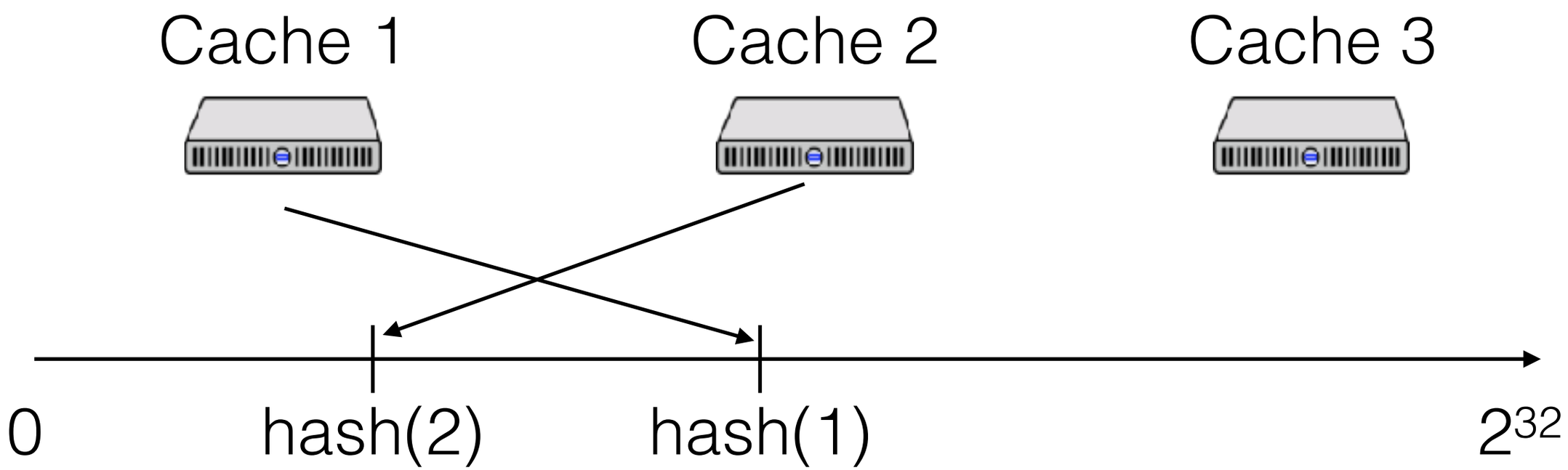
Proposal 3

First, hash the node ids



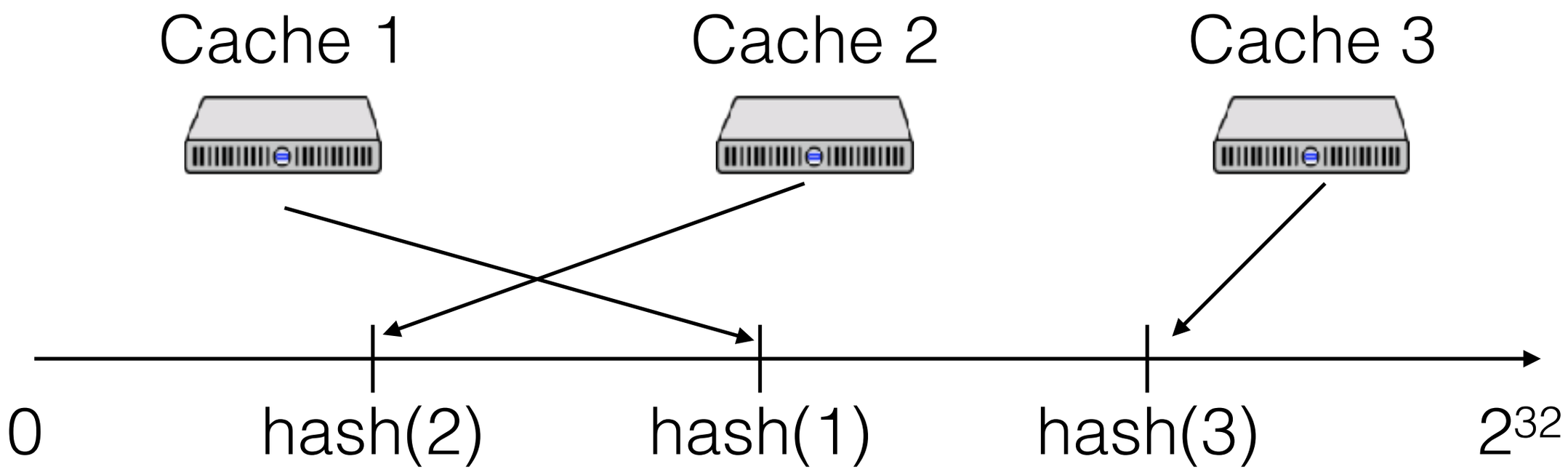
Proposal 3

First, hash the node ids



Proposal 3

First, hash the node ids



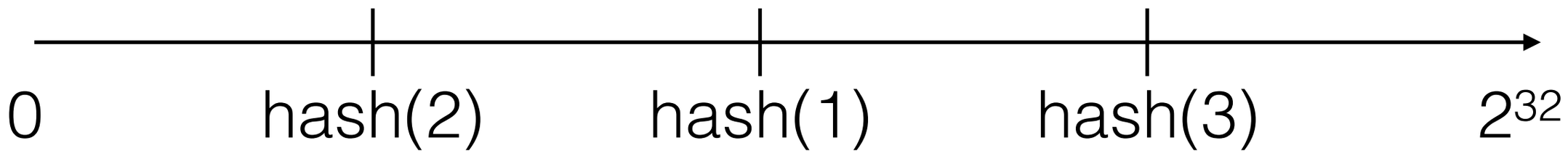
Proposal 3

First, hash the node ids

Cache 1

Cache 2

Cache 3



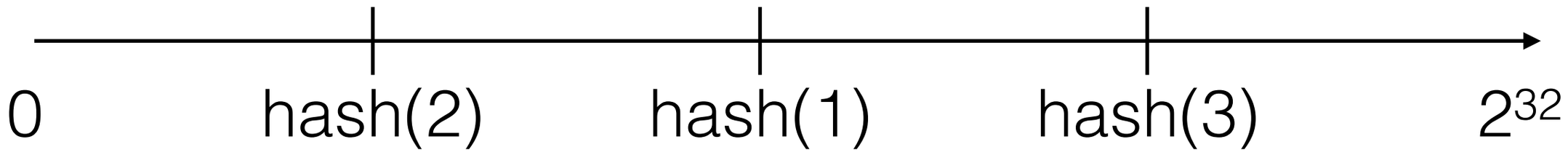
Proposal 3

First, hash the node ids

Cache 1 

Cache 2 

Cache 3 



Keys are hashed, then go to the “next” node

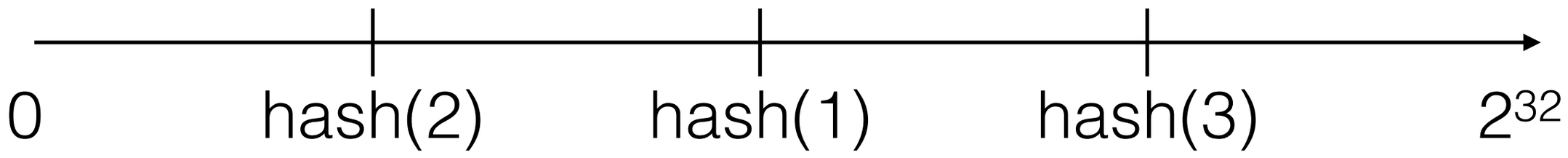
Proposal 3

First, hash the node ids

Cache 1 

Cache 2 

Cache 3 

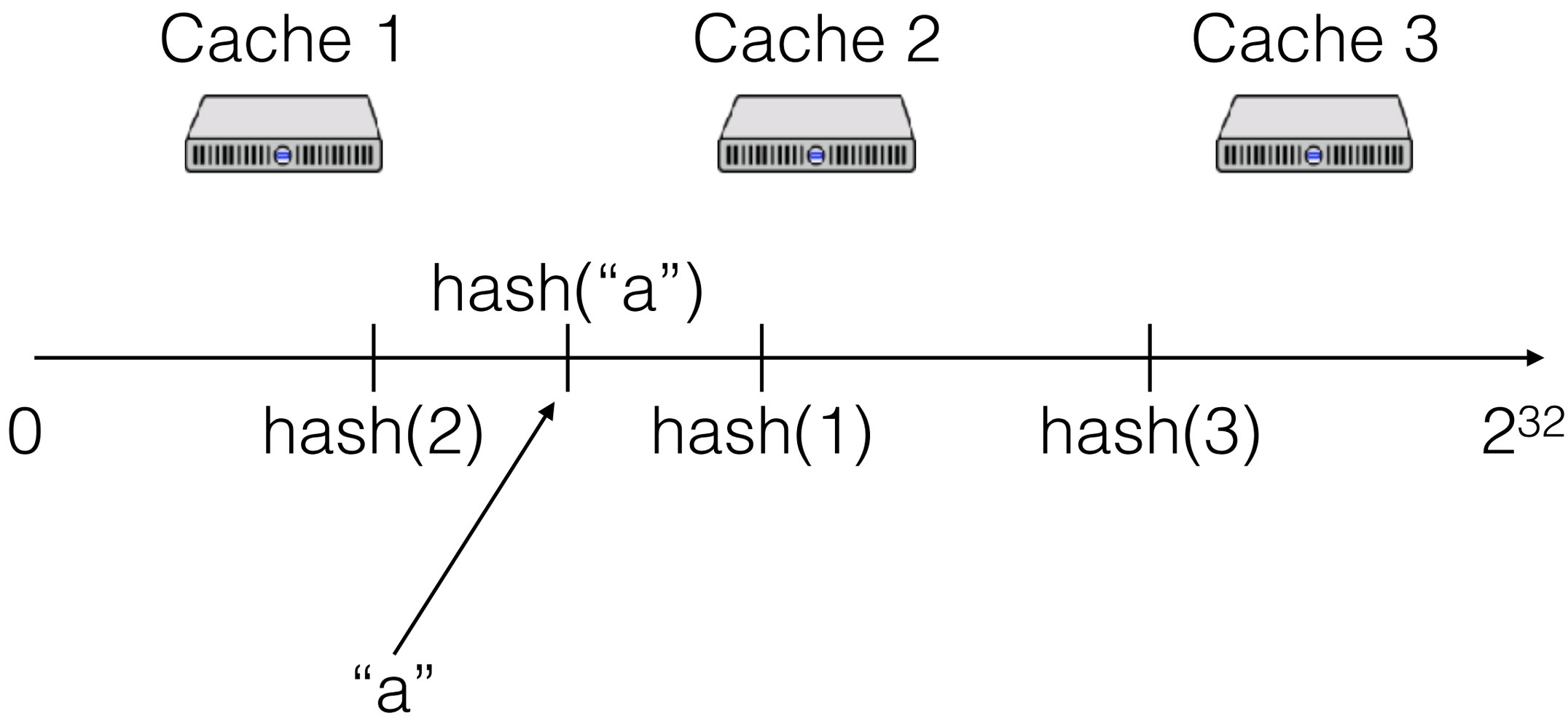


“a”

Keys are hashed, then go to the “next” node

Proposal 3

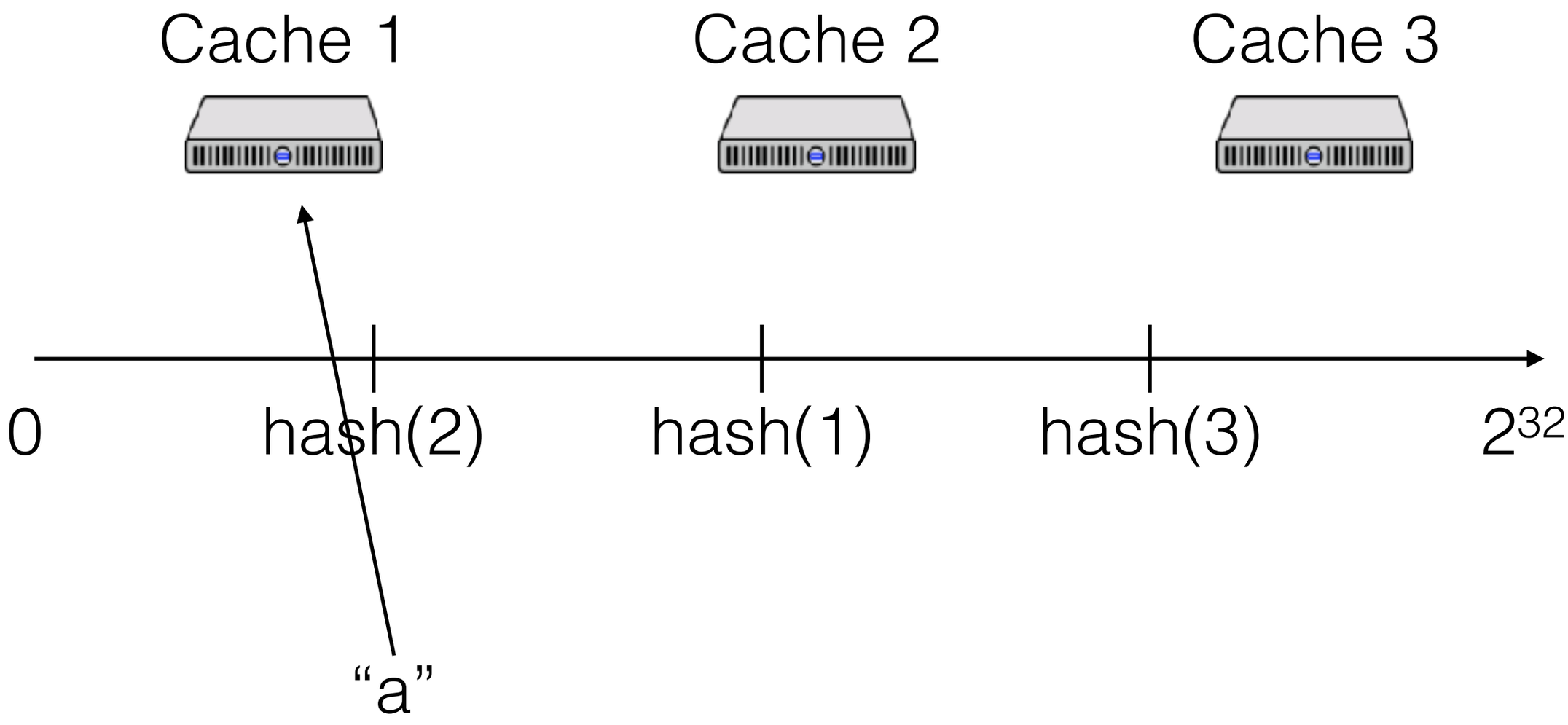
First, hash the node ids



Keys are hashed, then go to the “next” node

Proposal 3

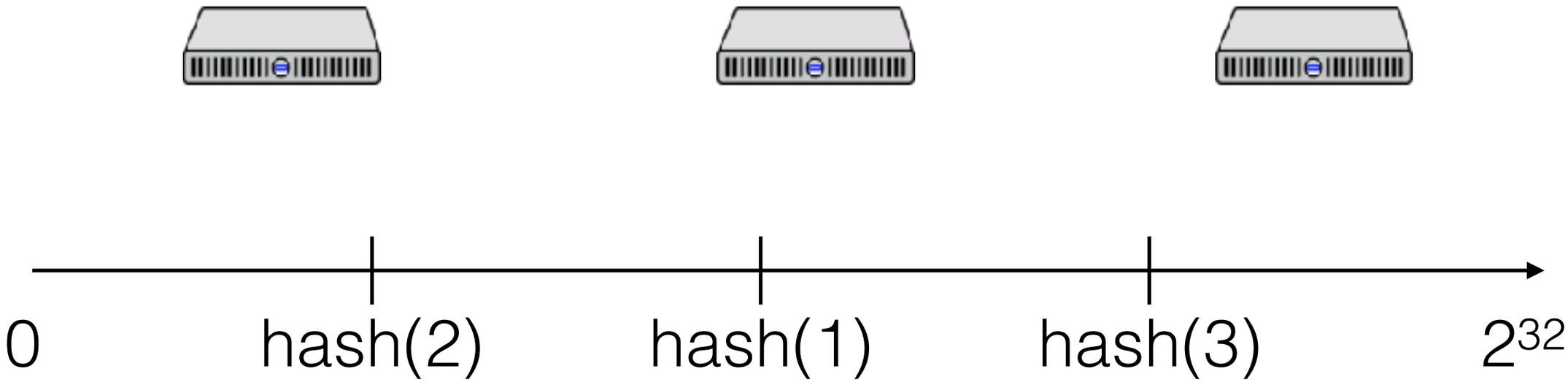
First, hash the node ids



Keys are hashed, then go to the “next” node

Proposal 3

First, hash the node ids

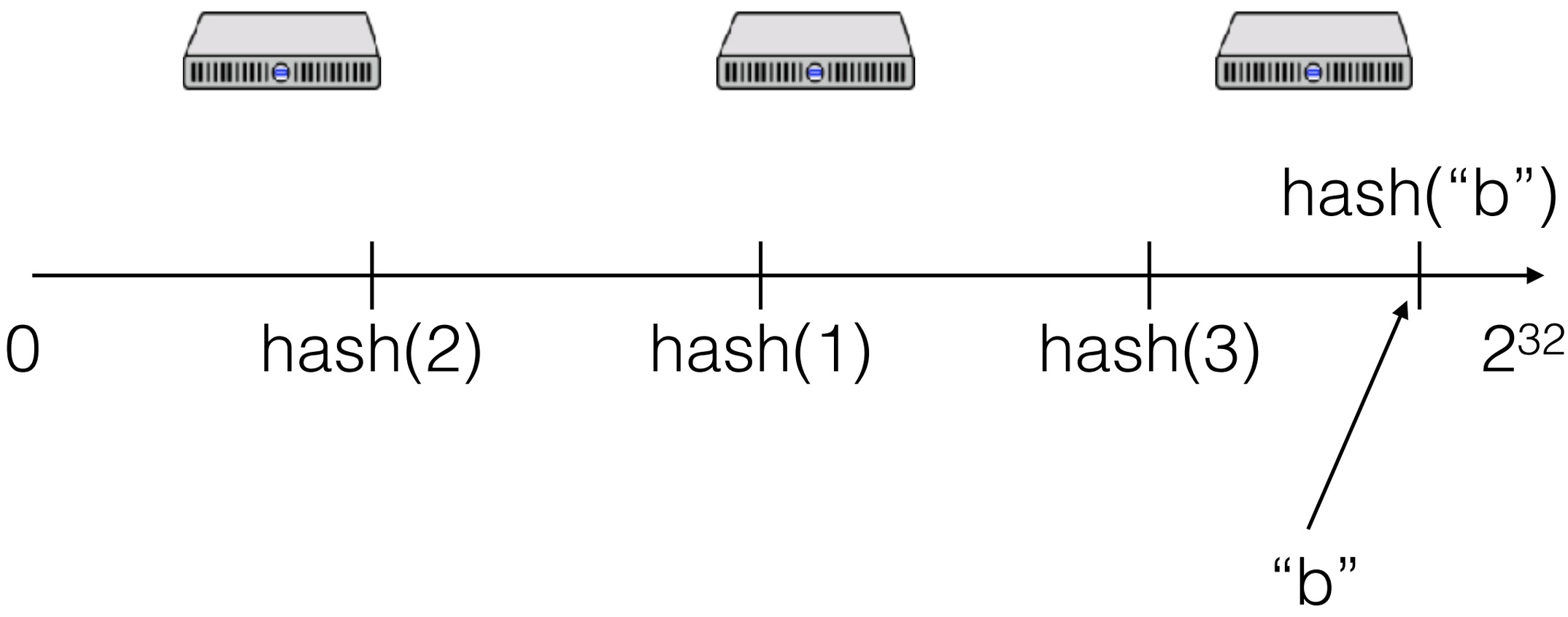


“b”

Keys are hashed, then go to the “next” node

Proposal 3

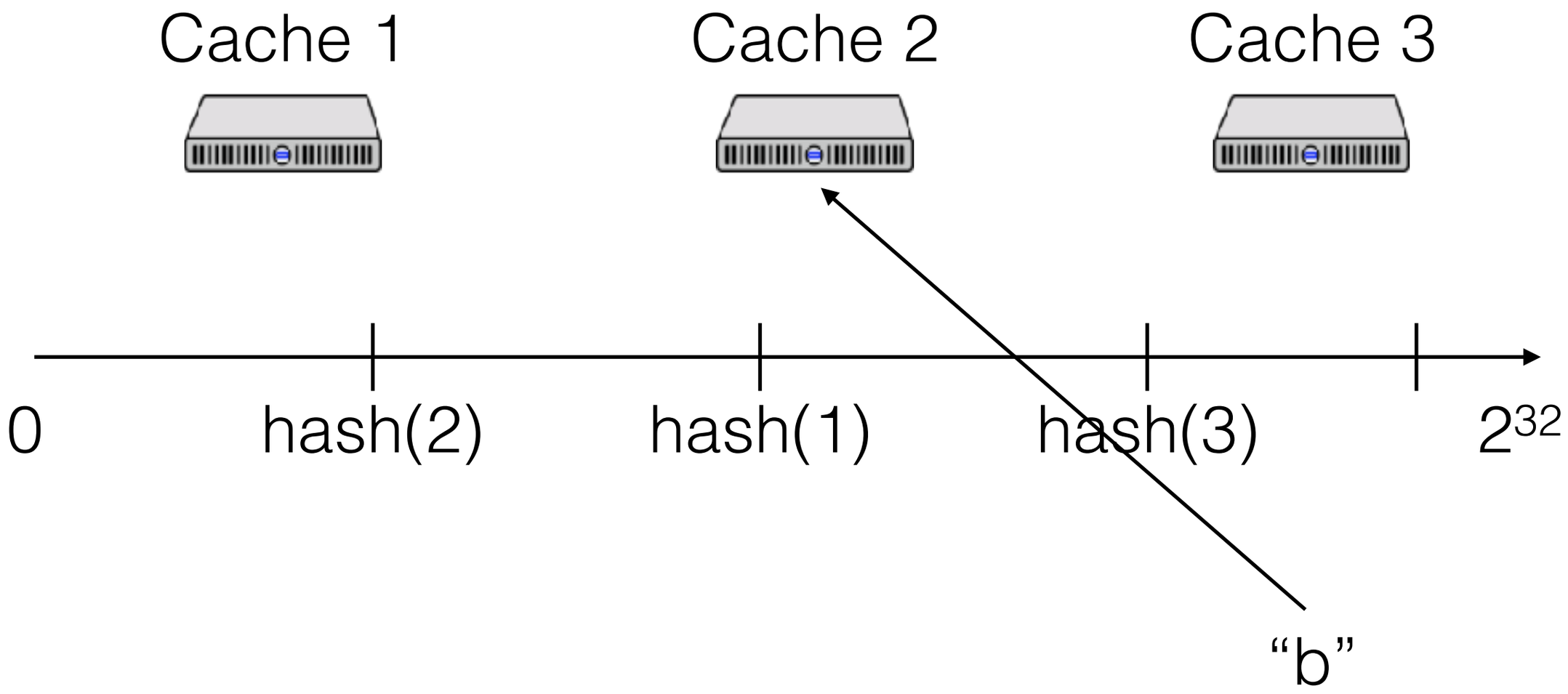
First, hash the node ids



Keys are hashed, then go to the “next” node

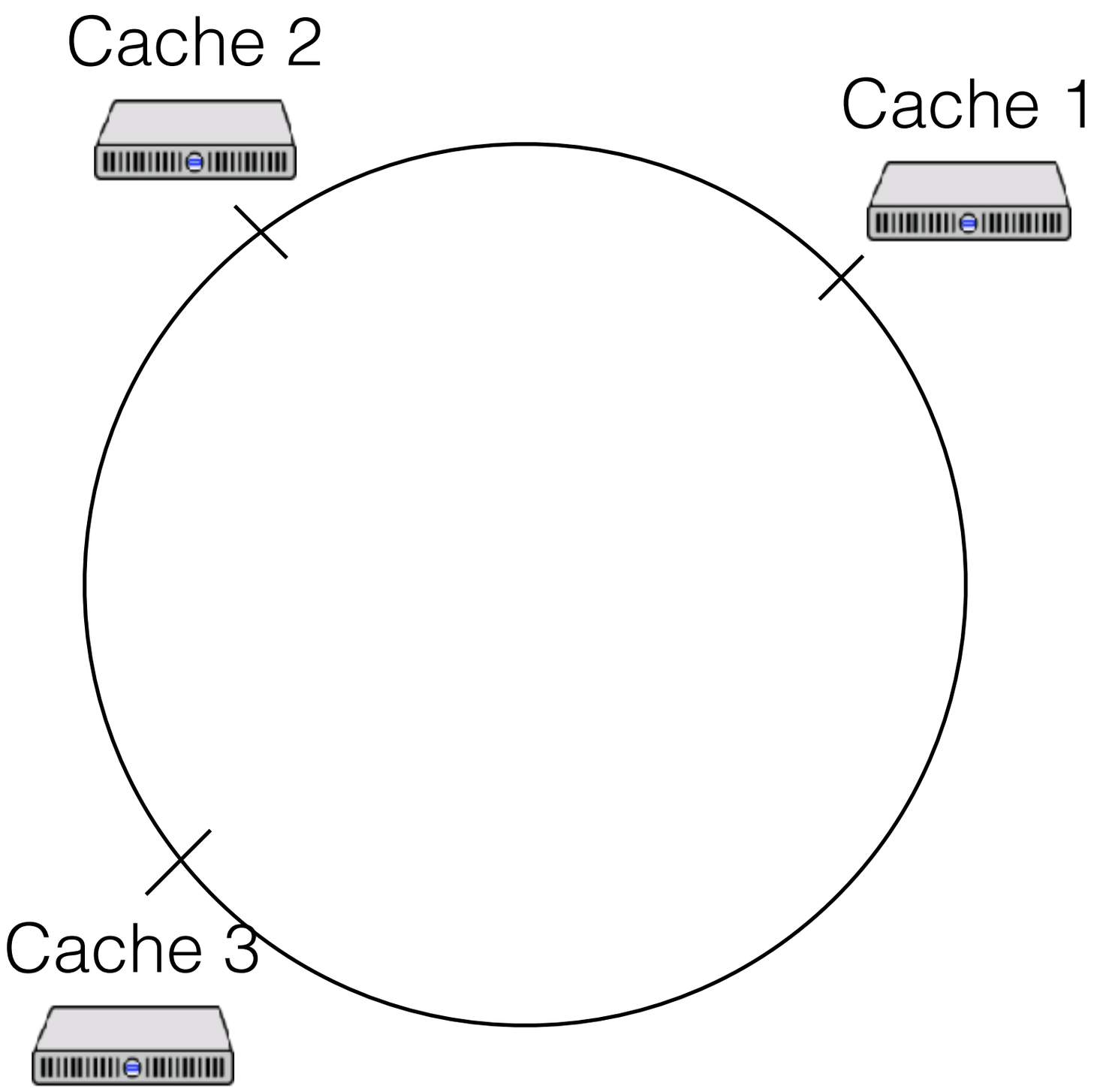
Proposal 3

First, hash the node ids

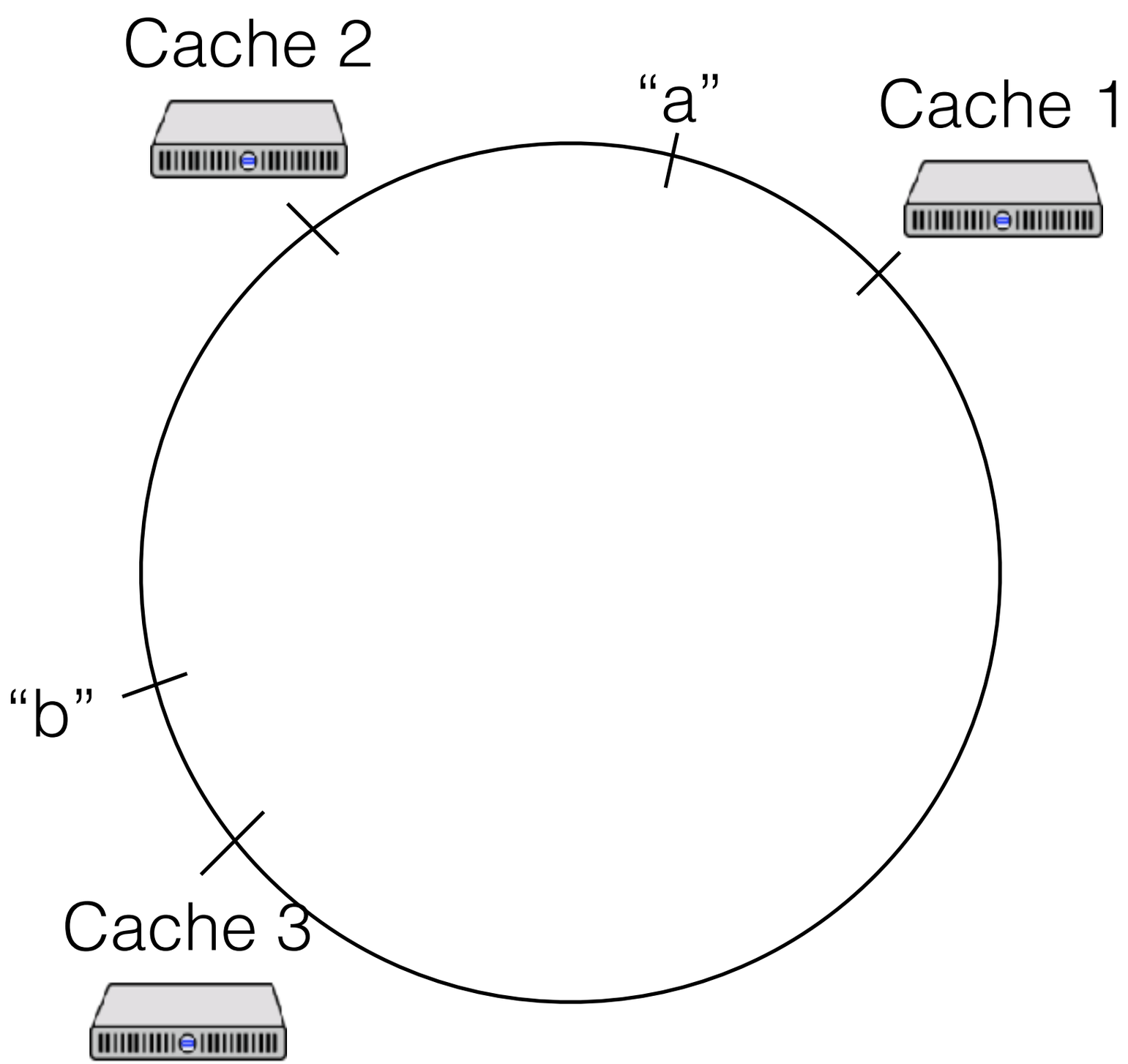


Keys are hashed, then go to the "next" node

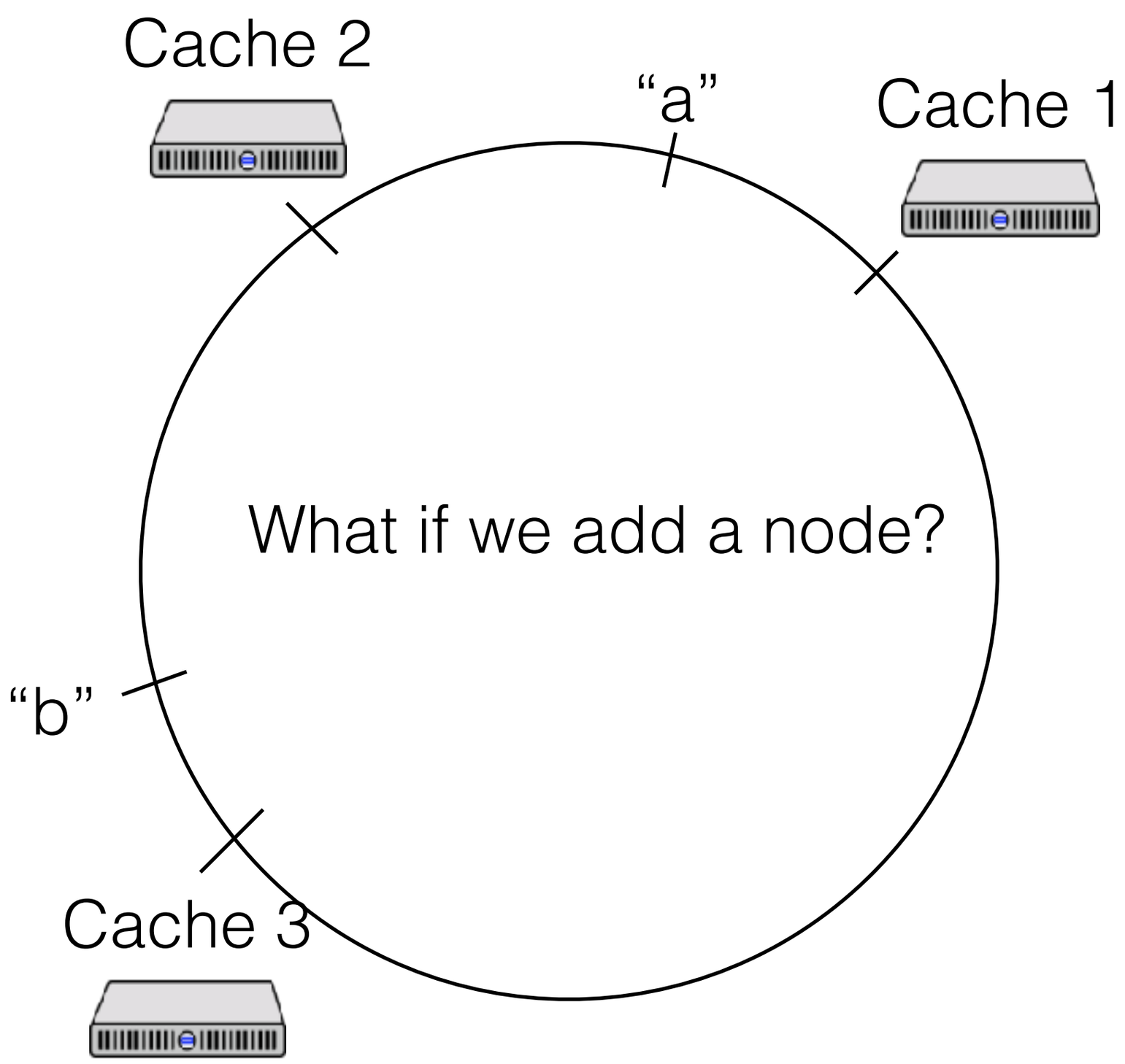
Proposal 3



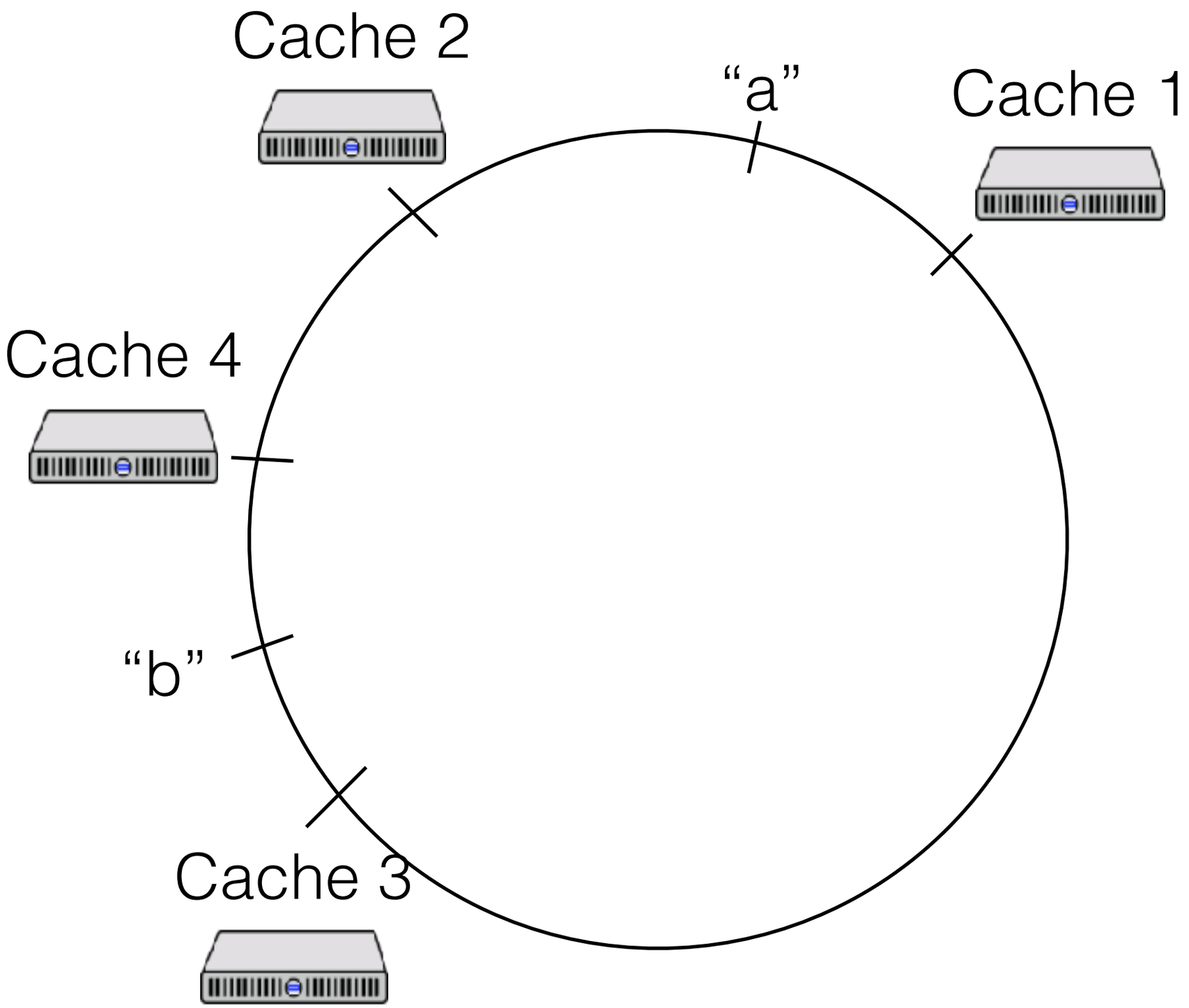
Proposal 3



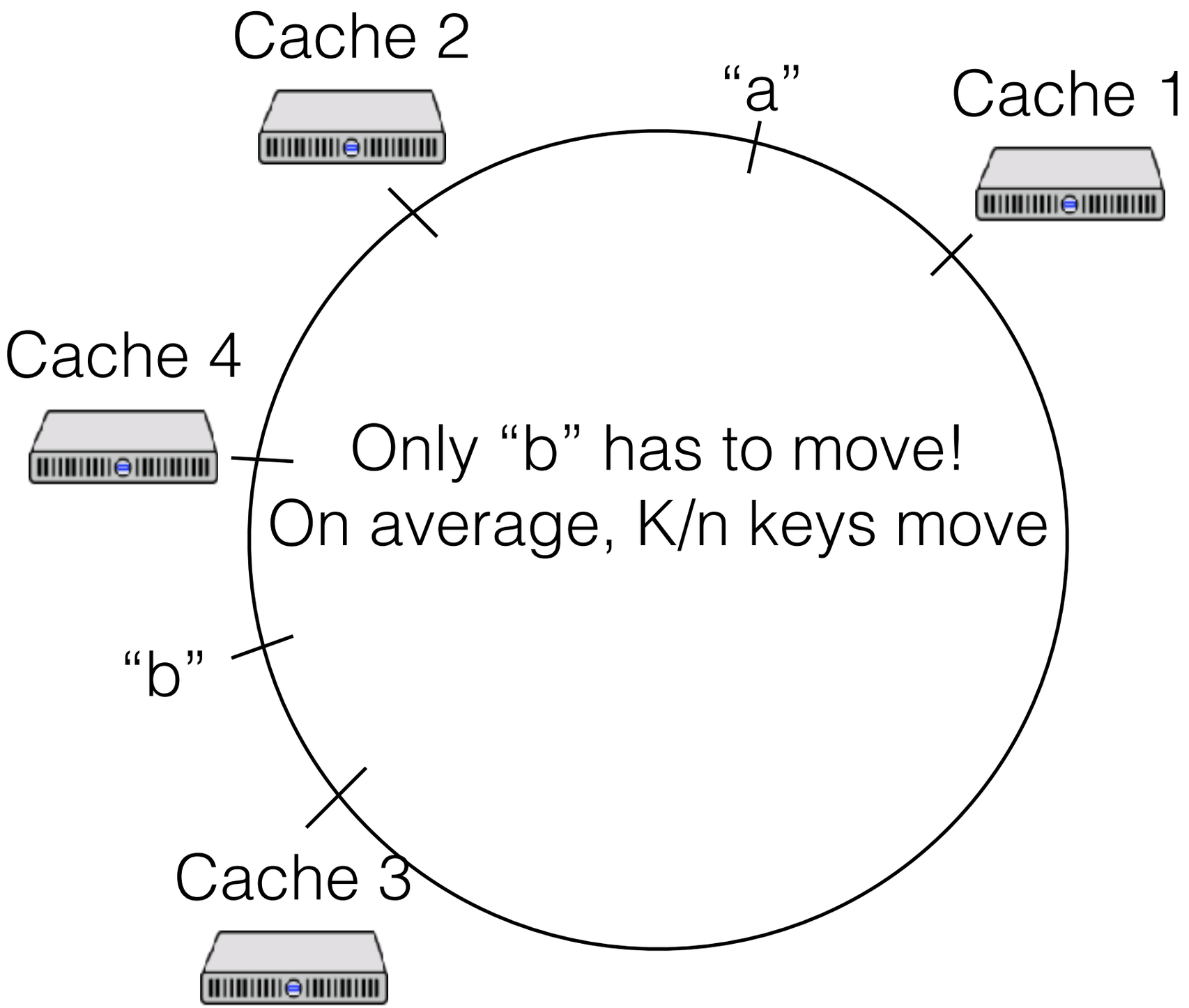
Proposal 3



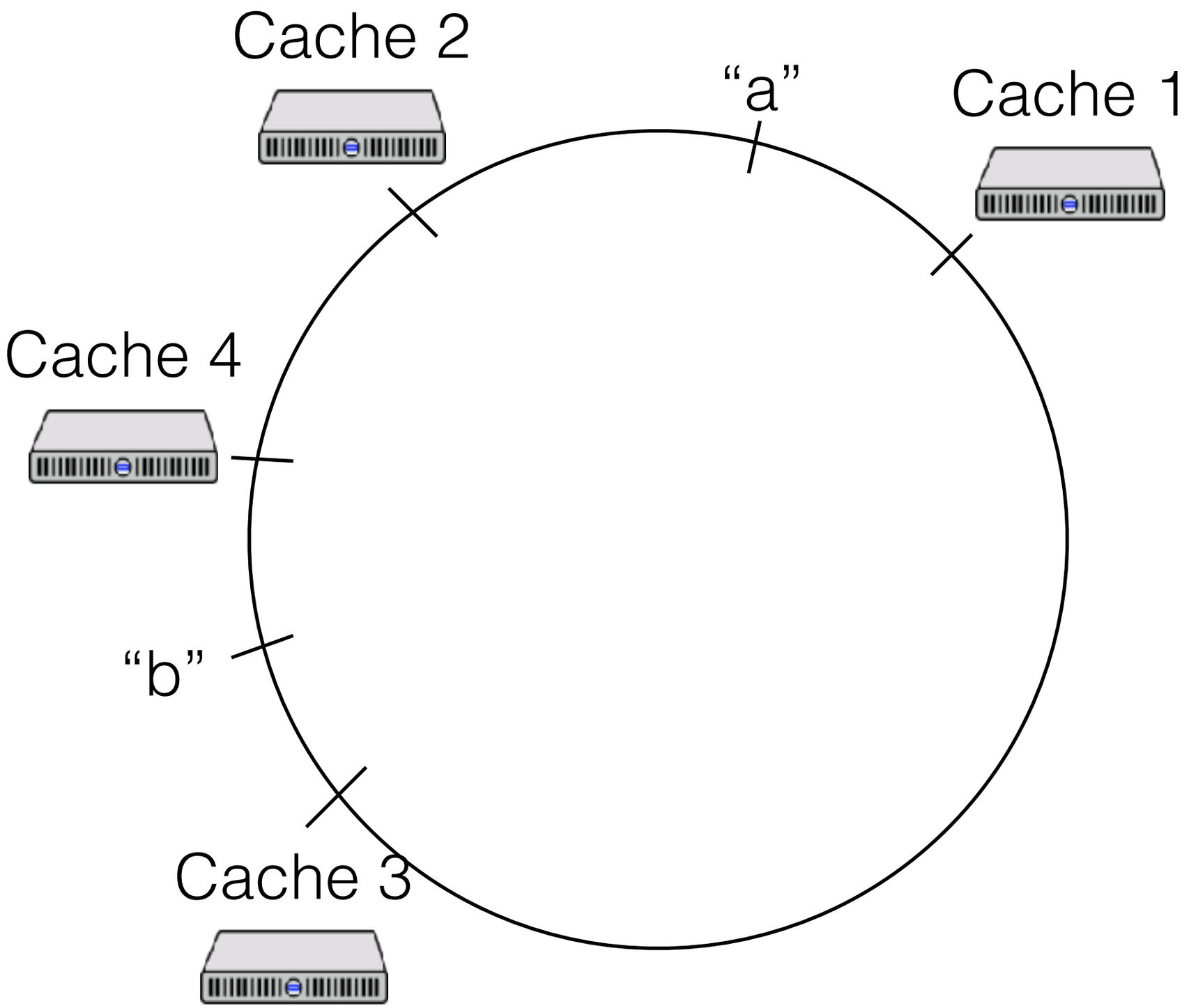
Proposal 3



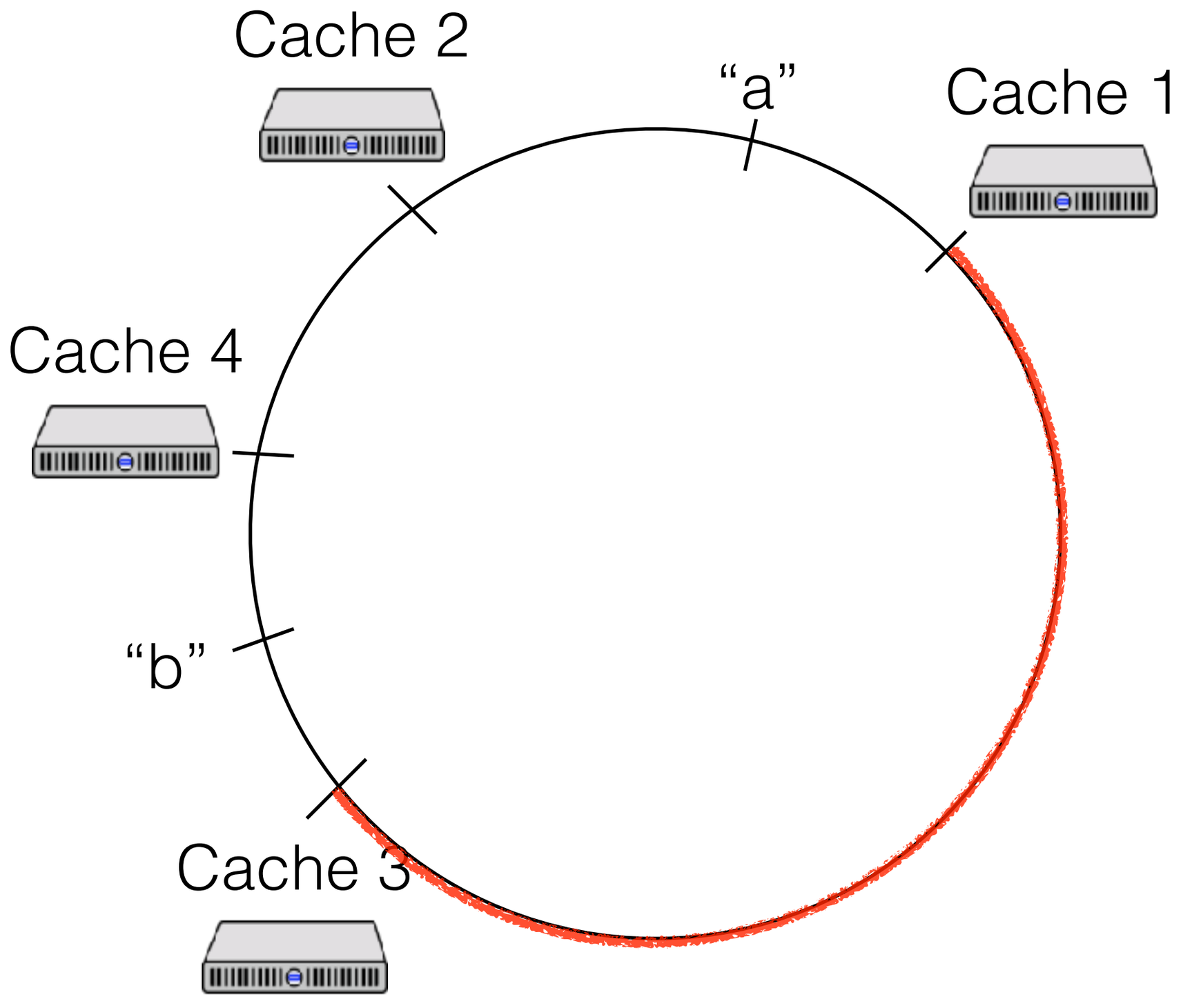
Proposal 3


Cache 2
“a”
Cache 1
Cache 4
Only “b” has to move!
On average, K/n keys move
“b”
Cache 3

Proposal 3



Proposal 3



Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Requirement 3: can add/remove nodes w/o redistributing too many keys

Proposal 4

First, hash the node ids to *multiple locations*

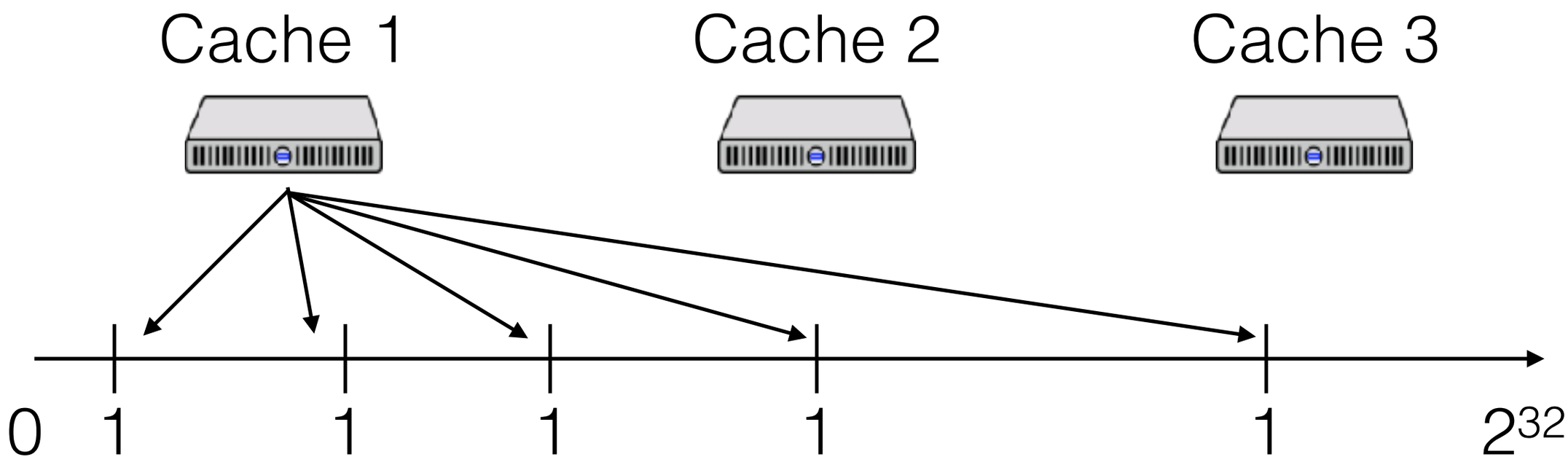
Cache 1

Cache 2

Cache 3

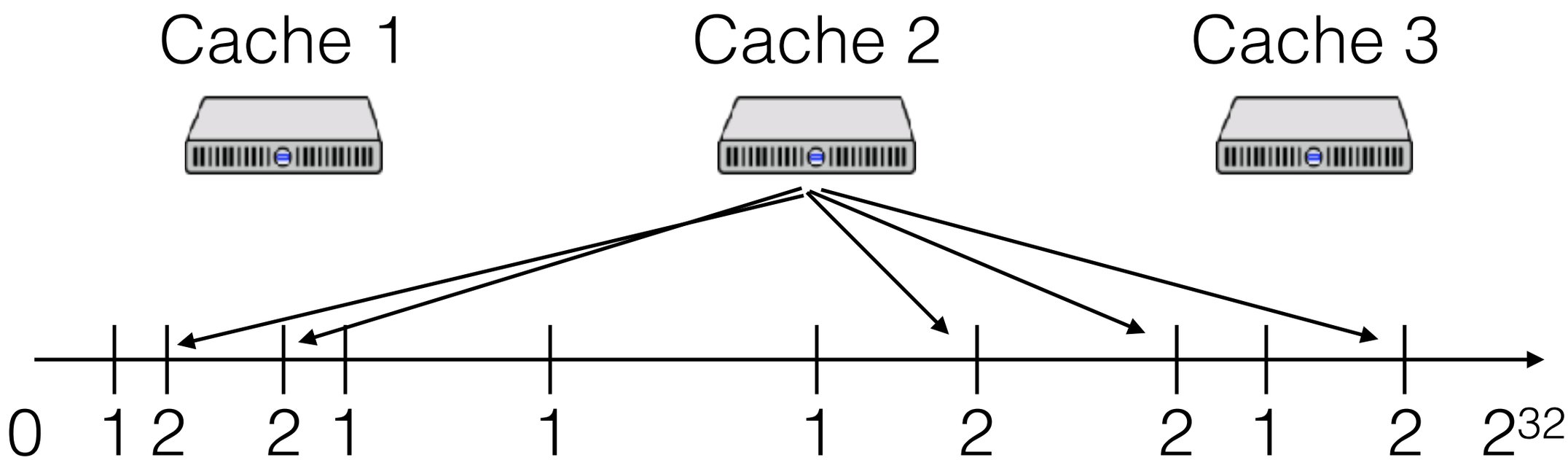
Proposal 4

First, hash the node ids to *multiple locations*



Proposal 4

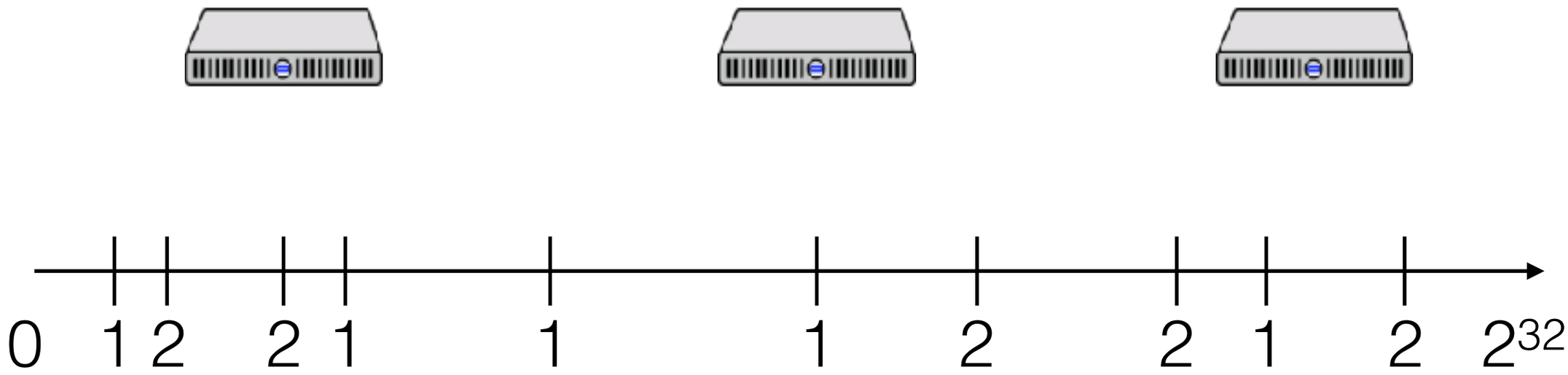
First, hash the node ids to *multiple locations*



Proposal 4

First, hash the node ids to *multiple locations*

Cache 1 Cache 2 Cache 3



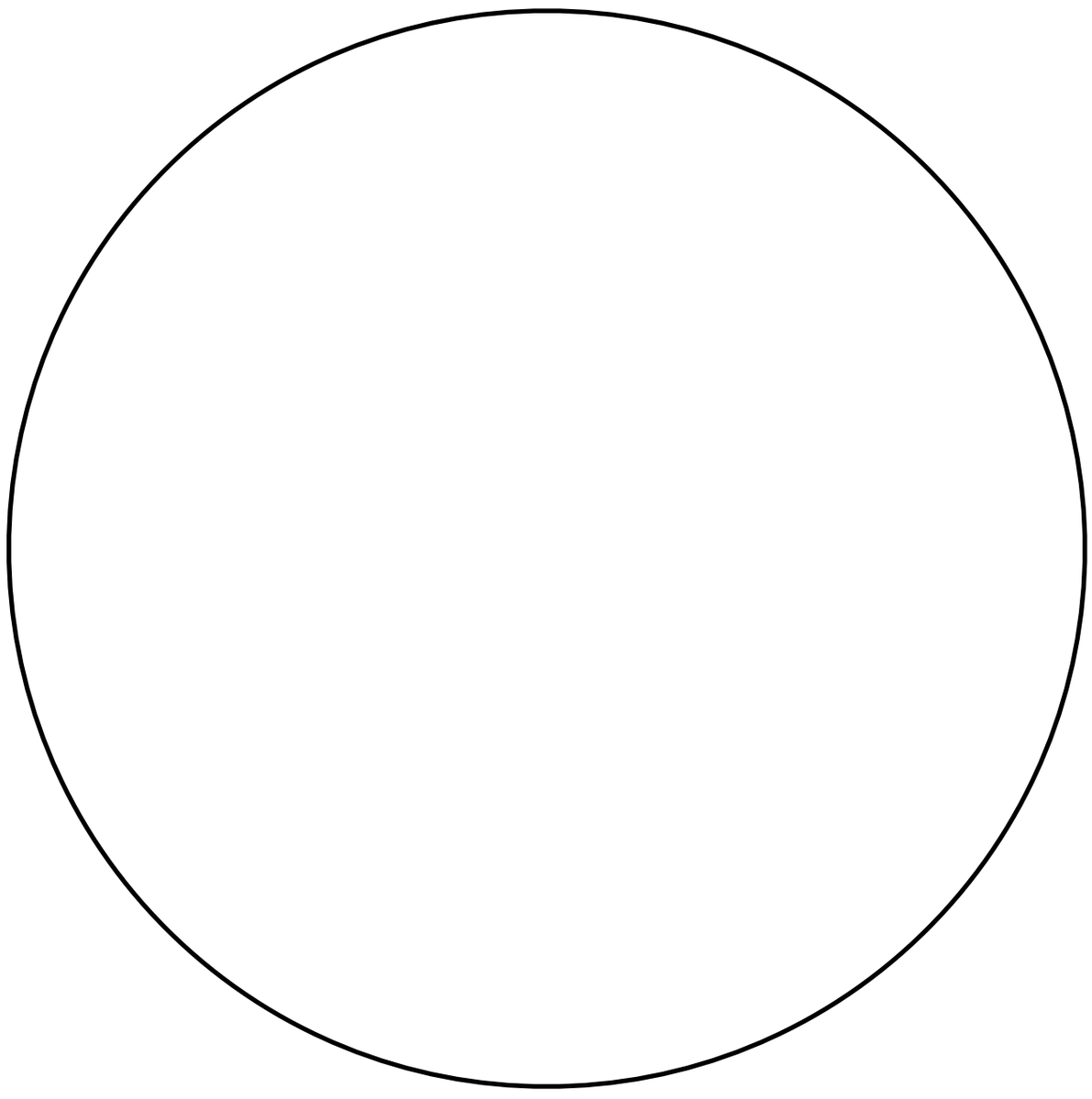
As it turns out, hash functions come in families s.t. their members are independent. So this is easy!

Proposal 4

Cache 1

Cache 2

Cache 3

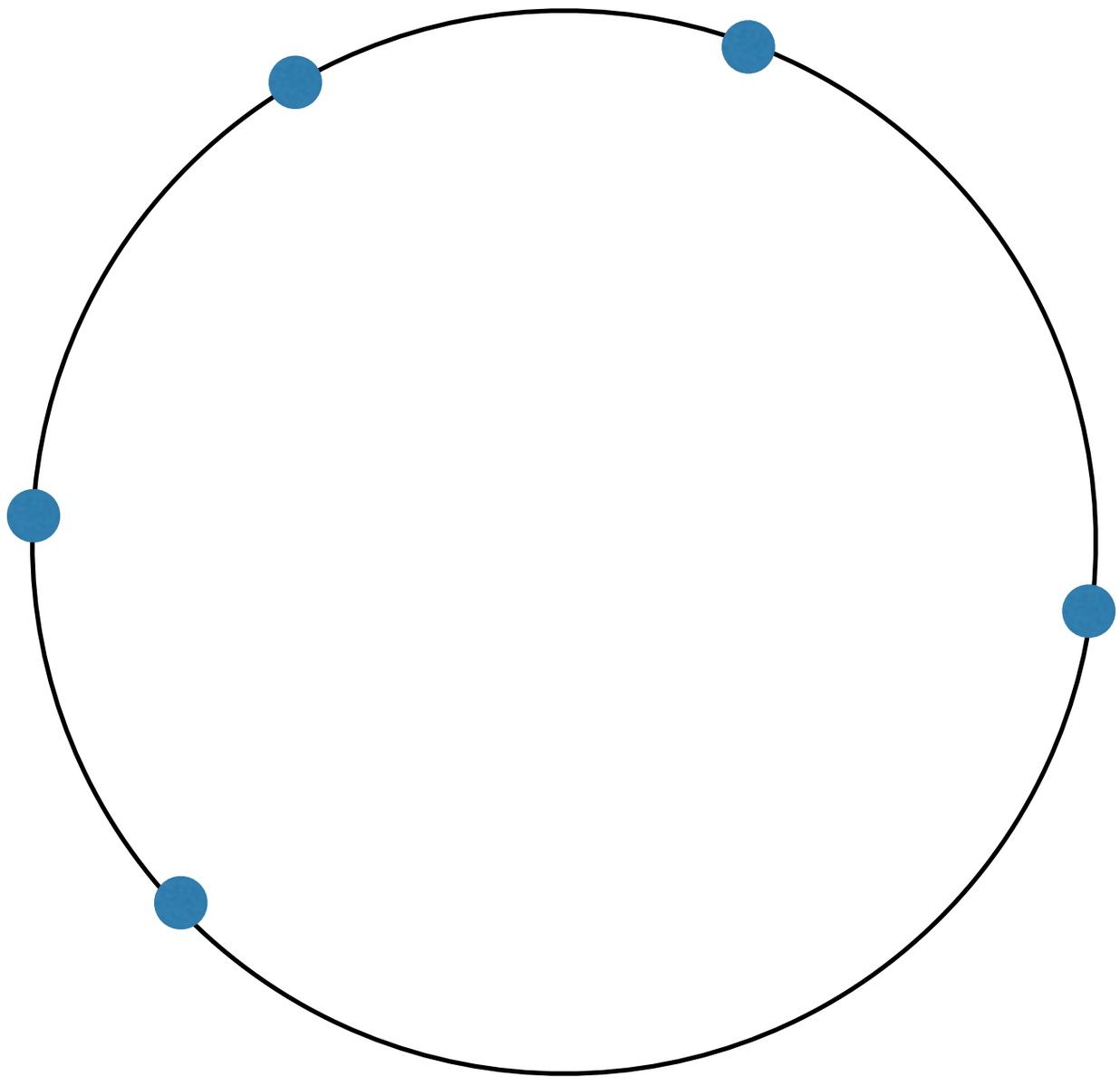


Proposal 4

Cache 1

Cache 2

Cache 3

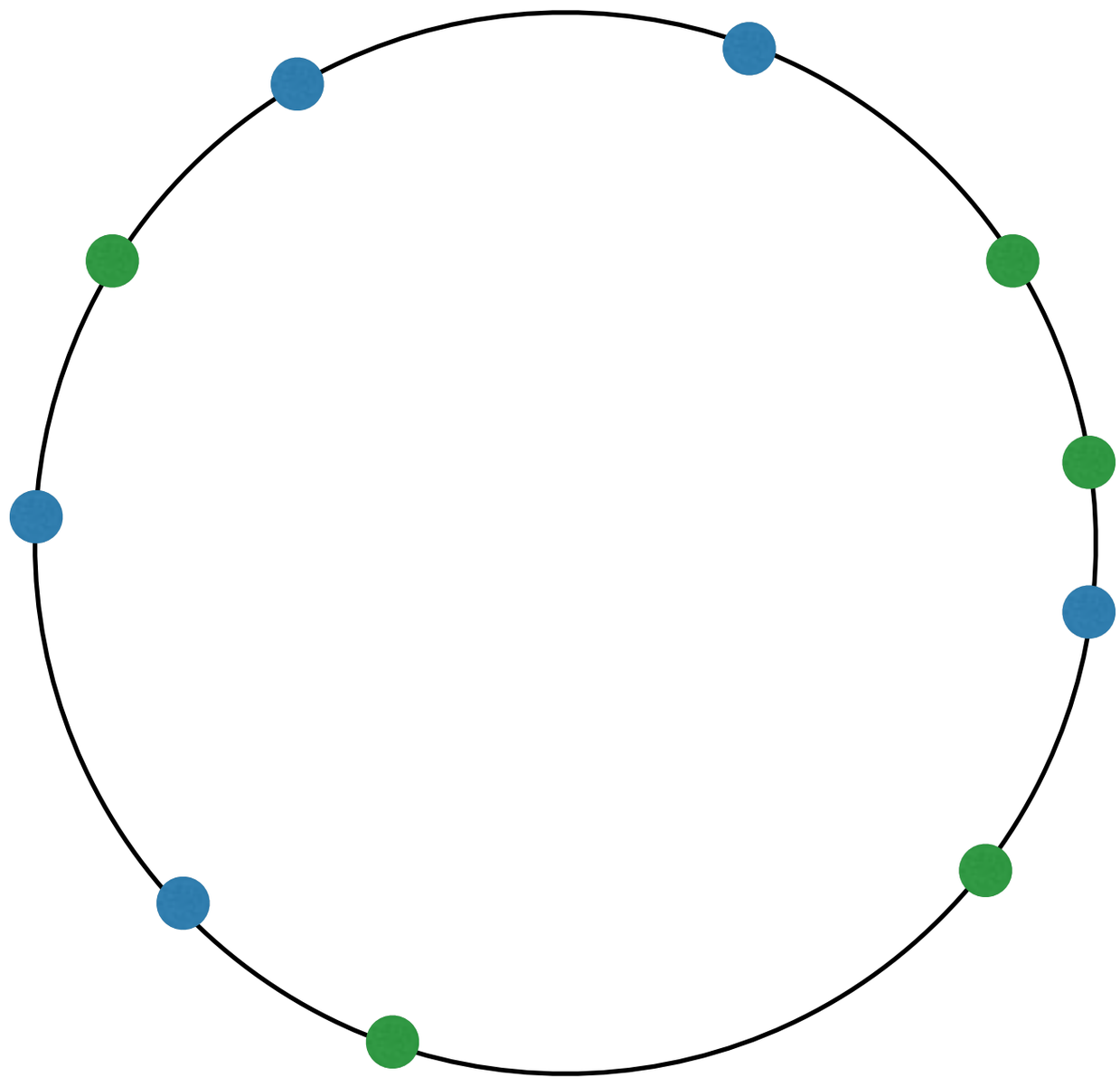


Proposal 4

Cache 1

Cache 2

Cache 3

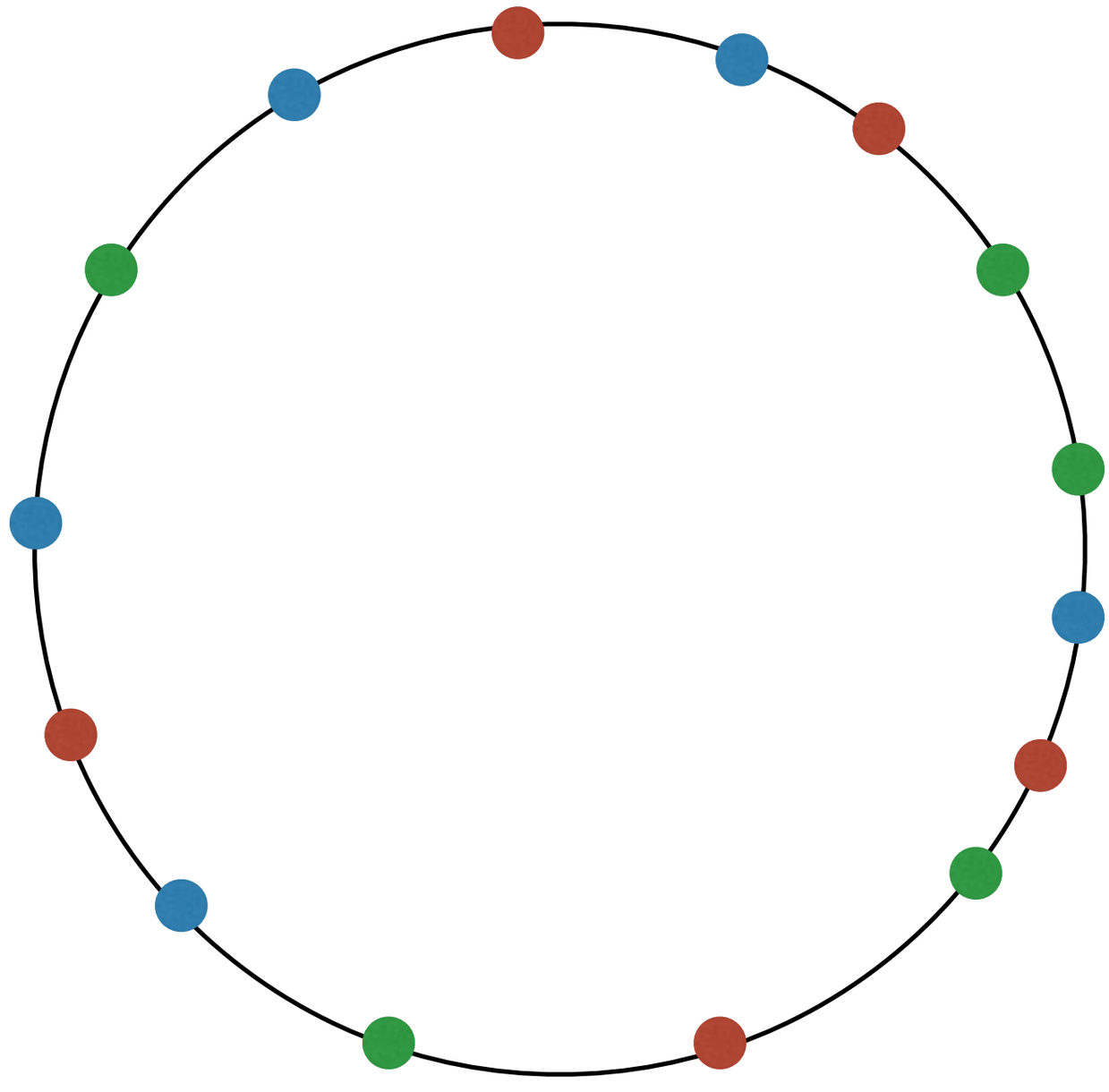


Proposal 4

Cache 1

Cache 2

Cache 3



Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Requirement 3: can add/remove nodes w/o redistributing too many keys

Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Requirement 3: can add/remove nodes w/o redistributing too many keys

Recap: consistent hashing

Node ids hashed to many pseudorandom points on a circle

Keys hashed onto circle, assigned to “next” node

Idea used widely:

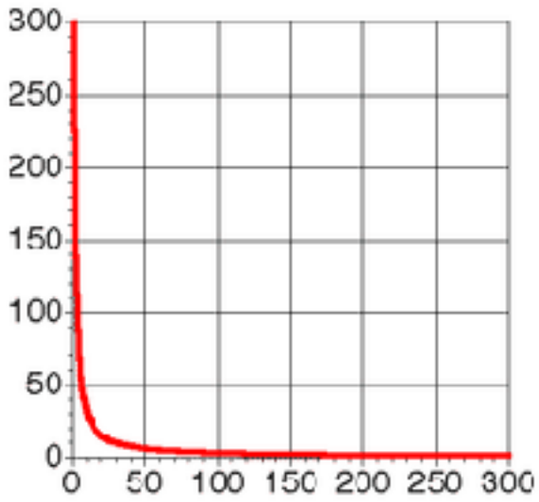
- Developed for Akamai CDN
- Used in Chord distributed hash table
- Used in Dynamo distributed DB

Further issues in load balancing

Accesses to keys are often Zipf-distributed

- Most popular accessed twice as often as 2nd-most

- Accessed 3x as often as 3rd-most, etc.



How to load balance “hot” keys?

Next week

Start of 3 weeks on “distributed systems in practice”

Lots of papers and discussion

Next week

Monday: Yegge on Service-Oriented Architectures

- Steve Yegge, prolific programmer and blogger

- Moved from Amazon to Google

- Monday's reading is an accidentally-leaked memo about differences between Amazon's and Google's system architectures (at that time)

- Advocates for SOA: separating applications (e.g. Google Search, Amazon) into many primitive services, run internally as products