Consistent Hashing

Tom Anderson and Doug Woos

Scaling Paxos: Shards

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

- all-to-all communication among servers on each op

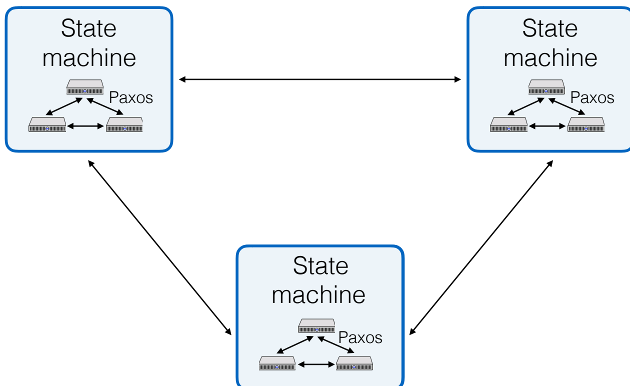
What if we want to scale to more clients?

Sharding: assign a subset of keys to each Paxos group

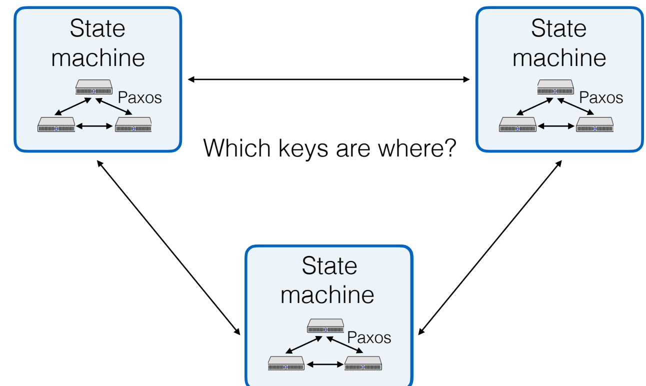
Recall: linearizable if

- clients do their operations in order (if needed)
- servers linearize each key

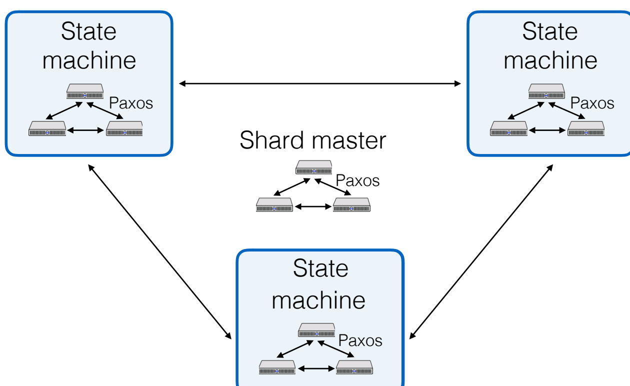
Replicated, Sharded Database



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

GET index.html

Client

Another scenario

index.html

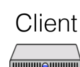
Client

Another scenario

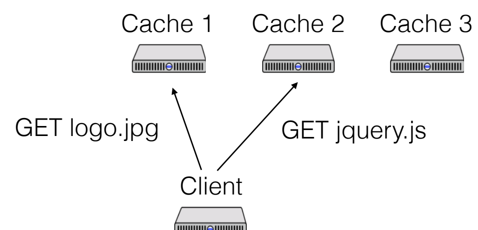
index.html

Links to: logo.jpg, jquery.js, …

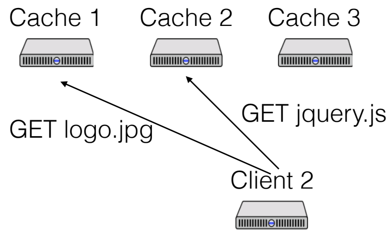
Client



Another scenario



Another scenario



Other Examples

Scalable shopping cart service

Scalable email service

Scalable cache layer (Memcache)

Scalable network path allocation

Scalable network function virtualization (NFV)

…

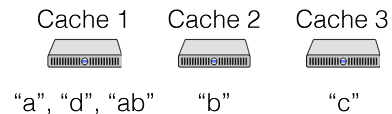
What's in common?

Want to assign keys to servers w/o communication

Requirement 1: clients all have same assignment

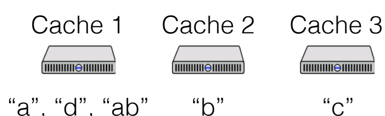
Proposal 1

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



Proposal 1

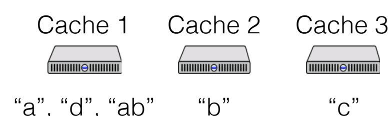
For n nodes, a key k goes to k mod n



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

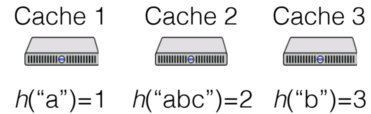
Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys uniformly distributed

Proposal 2: Hashing

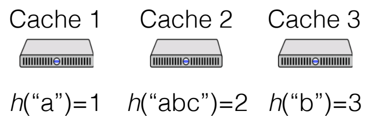
For *n* nodes, a key *k* goes to *hash(k)* mod *n*



Hash distributes keys uniformly

Proposal 2: Hashing

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

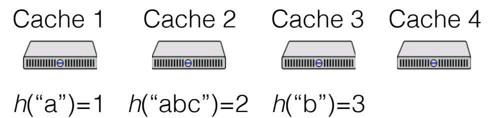


Hash distributes keys uniformly

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

Proposal 2: Hashing

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

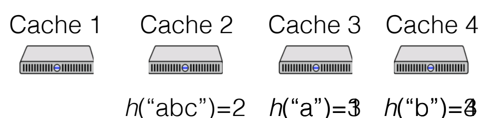


Hash distributes keys uniformly

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

Proposal 2: Hashing

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

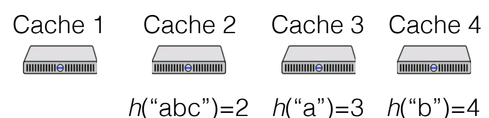


Hash distributes keys uniformly

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

Proposal 2: Hashing

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



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

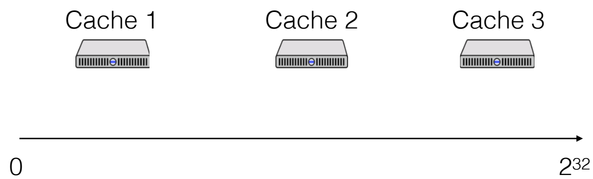
Requirement 3: can add/remove nodes w/o redistributing too many keys

Proposal 3: Consistent Hashing

First, hash the node ids

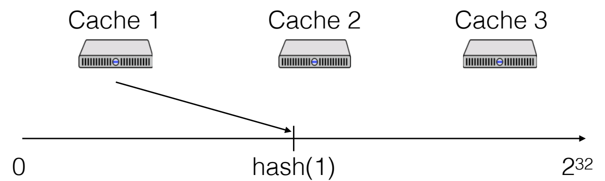
Proposal 3: Consistent Hashing

First, hash the node ids



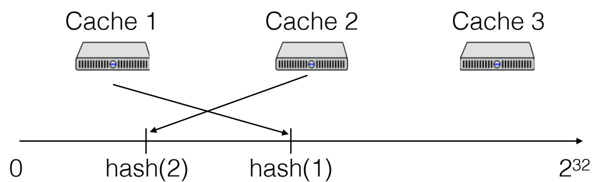
Proposal 3: Consistent Hashing

First, hash the node ids



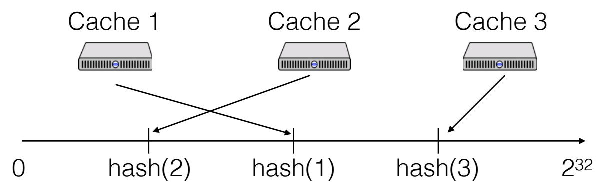
Proposal 3: Consistent Hashing

First, hash the node ids



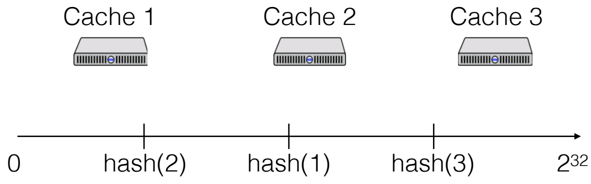
Proposal 3: Consistent Hashing

First, hash the node ids



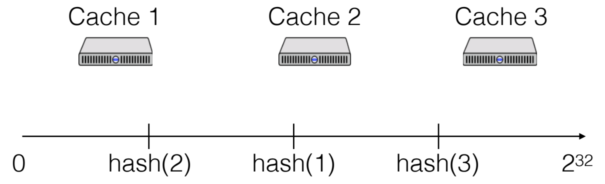
Proposal 3: Consistent Hashing

First, hash the node ids



Proposal 3: Consistent Hashing

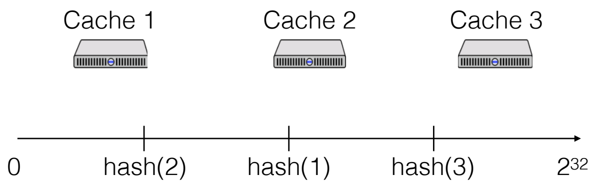
First, hash the node ids



Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

First, hash the node ids

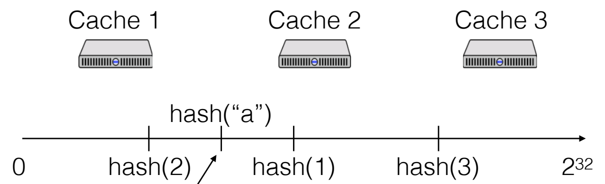


“a”

Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

First, hash the node ids

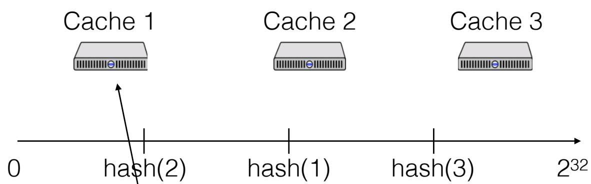


“a”

Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

First, hash the node ids

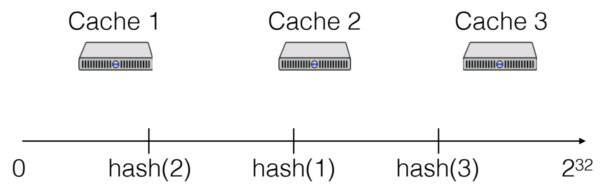


“a”

Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

First, hash the node ids

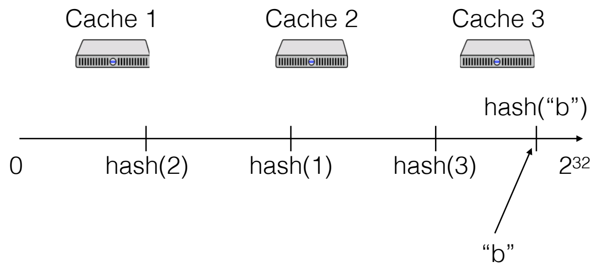


“b”

Keys are hashed, go to the “next” node

Proposal 3: Consistent Hashing

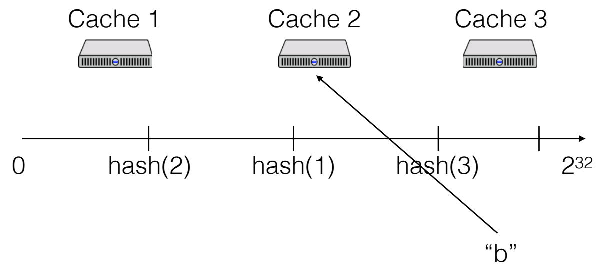
First, hash the node ids



Keys are hashed, go to the “next” node

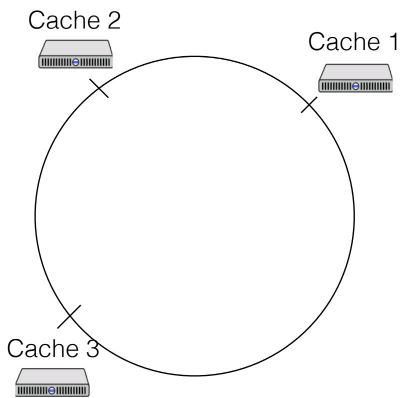
Proposal 3: Consistent Hashing

First, hash the node ids

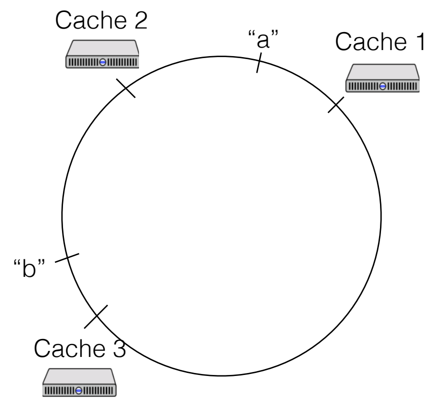


Keys are hashed, go to the “next” node

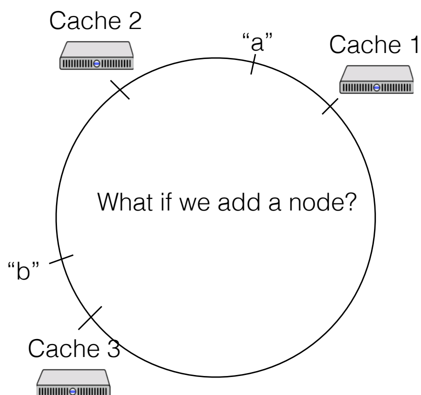
Proposal 3: Consistent Hashing



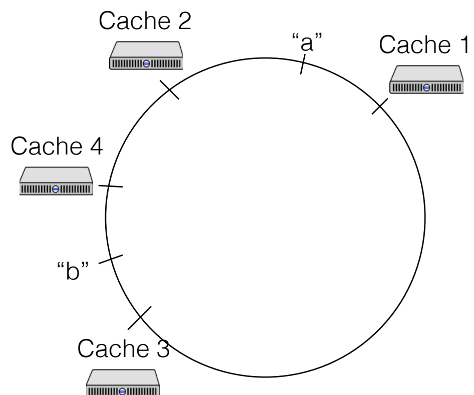
Proposal 3: Consistent Hashing



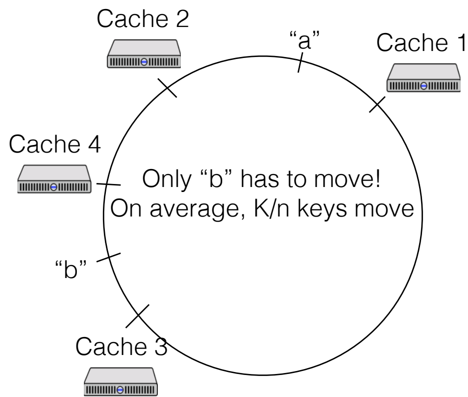
Proposal 3: Consistent Hashing



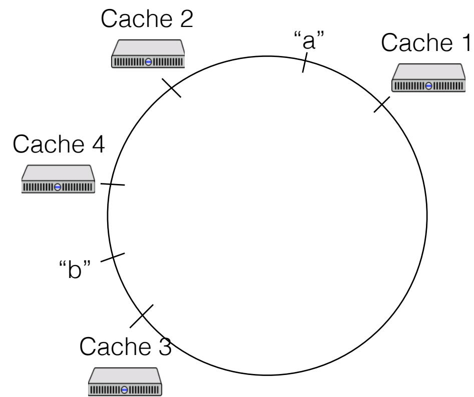
Proposal 3: Consistent Hashing



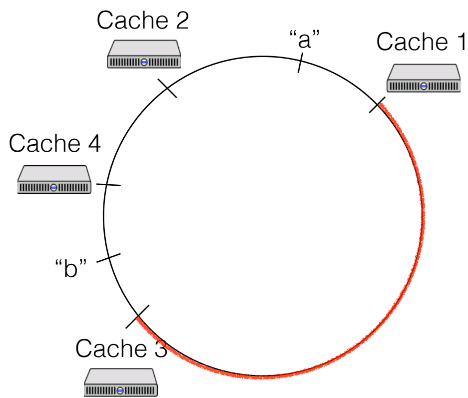
Proposal 3: Consistent Hashing



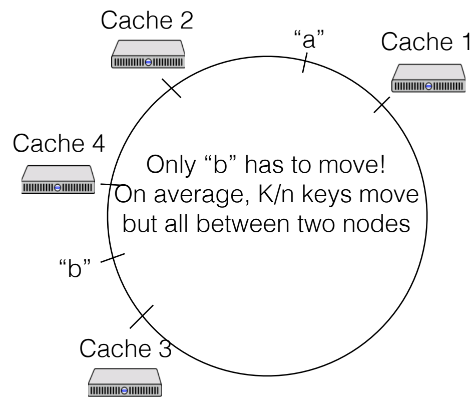
Proposal 3: Consistent Hashing



Proposal 3: Consistent Hashing



Proposal 3: Consistent Hashing



Requirements, revisited

Requirement 1: clients all have same assignment

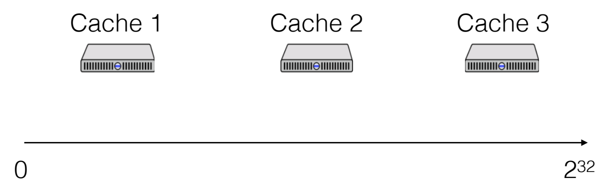
Requirement 2: keys evenly distributed

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

Requirement 4: parcel out work of redistributing keys

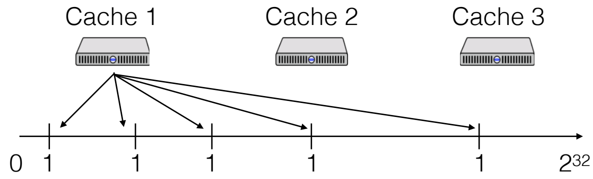
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



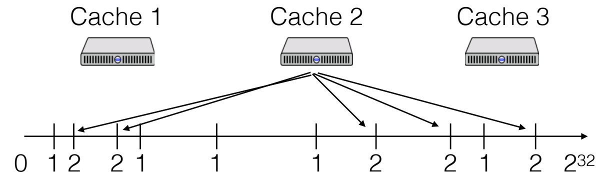
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



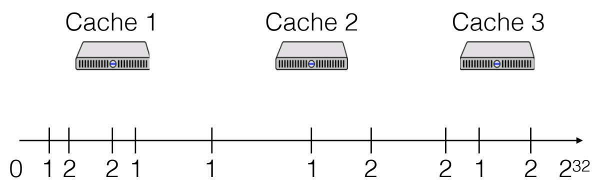
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*



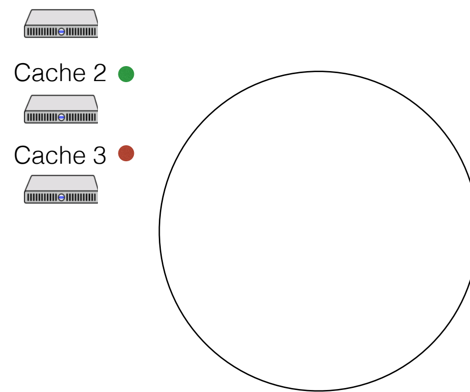
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*

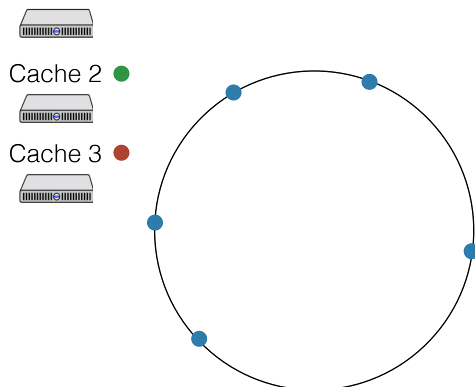


As it turns out, hash functions come in families s.t. their members are independent. So this is easy!

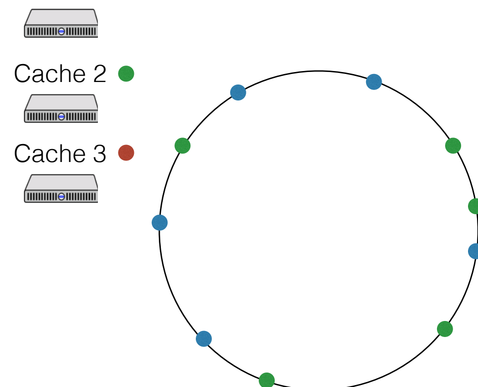
Prop 4: Virtual Nodes

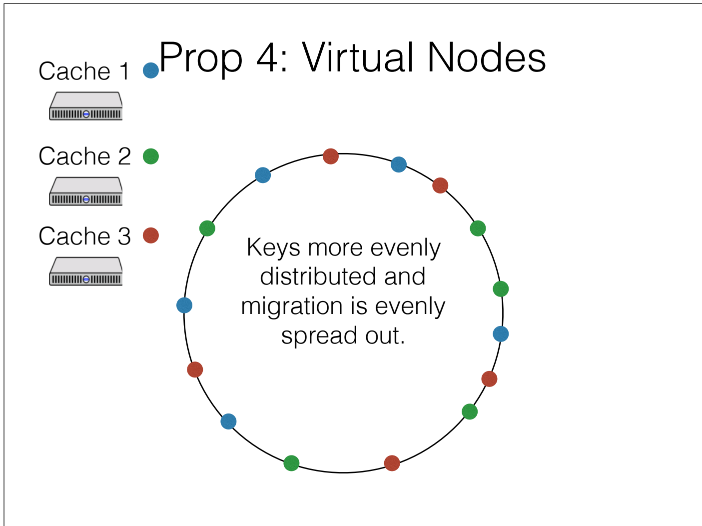


Prop 4: Virtual Nodes



Prop 4: Virtual Nodes





Requirements, revisited

Requirement 1: clients all have same assignment

Requirement 2: keys evenly distributed

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

Requirement 4: parcel out work of redistributing keys

Load Balancing At Scale

Suppose you have N servers

Using consistent hashing with virtual nodes:

- heaviest server has x% more load than the average

- lightest server has x% less load than the average

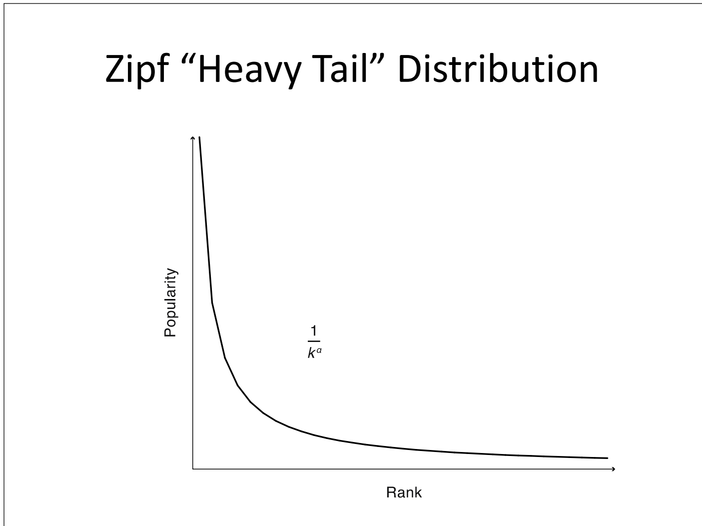
What is peak load of the system?

- N * load of average machine? No!

Need to minimize x

Key Popularity

- What if some keys are more popular than others
- Consistent 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 Examples

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

Whenever popularity is self-reinforcing

Proposal 5: Table Indirection

Consistent hashing is (mostly) stateless

- Given list of servers and # of virtual nodes, client can locate key
- Worst case unbalanced, especially with zipf

Add a small table on each client

- Table maps: virtual node -> server
- Shard master reassigns table entries to balance load

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

Next Week

Start of 3 weeks on “distributed systems in practice”

Lots of papers and discussion

Friday/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