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

State machine

Paxos

State machine

Paxos

State machine

Paxos
Replicated, Sharded Database

Which keys are where?
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
Another scenario

GET index.html
Another scenario

Google  Facebook  Netflix

index.html

Client
Another scenario

Google  Facebook  Netflix

index.html

Links to: logo.jpg, jquery.js, ...
Another scenario

GET logo.jpg

GET jquery.js

Cache 1

Cache 2

Cache 3

Client

Google

Facebook

NETFLIX
Another scenario

Google  Facebook  NETFLIX

Cache 1  Cache 2  Cache 3

GET logo.jpg  GET jquery.js

Client 2
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)
...

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 $k \mod n$

Cache 1: “a”, “d”, “ab”
Cache 2: “b”
Cache 3: “c”
Proposal 1

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

![Cache Diagram]

“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

Response Time $R$ vs. Utilization $U$

$R = S/(1-U)$

Variance in response time $\sim S/(1-U)^2$
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 $hash(k) \ mod \ n$

- $h(“a”) = 1$
- $h(“abc”) = 2$
- $h(“b”) = 3$

Hash distributes keys uniformly

But, new problem: what if we add a node?
Proposal 2: Hashing

For $n$ nodes, a key $k$ goes to $\text{hash}(k) \mod n$

Hash distributes keys uniformly

But, new problem: what if we add a node?
Proposal 2: Hashing

For $n$ nodes, a key $k$ goes to $\text{hash}(k) \mod n$

Hash distributes keys uniformly

But, new problem: what if we add a node?
Proposal 2: Hashing

For $n$ nodes, a key $k$ goes to $\text{hash}(k) \mod n$

$$h(\text{"abc"})=2 \quad h(\text{"a"})=3 \quad h(\text{"b"})=4$$

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

Cache 1  Cache 2  Cache 3

0  \(2^{32}\)
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

hash(1)
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1

Cache 2

Cache 3

0

hash(2)

hash(1)

$2^{32}$
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

0  hash(2)  hash(1)  hash(3)  $2^{32}$
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1

Cache 2

Cache 3

0  hash(2)  hash(1)  hash(3)  $2^{32}$
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

hash(1)  hash(2)  hash(3)

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

hash(1)  hash(2)  hash(3)

0  hash(2)  hash(1)  hash(3)  $2^{32}$

“a”

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

Cache 1

hash(1)

Cache 2

hash(2)

hash(3)

Cache 3

hash("a")

0

hash(2) hash(1) hash(3)

2^{32}

"a"

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1

Cache 2

Cache 3

0 hash(2) hash(1) hash(3) $2^{32}$

“b”

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  Cache 2  Cache 3

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

First, hash the node ids

Cache 1  

Cache 2  

Cache 3  

"b"

Keys are hashed, go to the “next” node
Proposal 3: Consistent Hashing

Cache 1

Cache 2

Cache 3
Proposal 3: Consistent Hashing
Proposal 3: Consistent Hashing

What if we add a node?
Proposal 3: Consistent Hashing

Cache 1

Cache 2

Cache 3

Cache 4

“a”

“b”
Proposal 3: Consistent Hashing

Only “b” has to move!
On average, K/n keys move
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

Only “b” has to move!
On average, K/n keys move but all between two nodes.
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*

```
Cache 1  Cache 2  Cache 3
```

0  \[2^{32}\]
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*

Cache 1 → Cache 2 → Cache 3

0 1 1 1 1 1 2^32
Proposal 4: Virtual Nodes

First, hash the node ids to *multiple locations*

Cache 1  
Cache 2  
Cache 3
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

Cache 1

Cache 2

Cache 3
Prop 4: Virtual Nodes
Prop 4: Virtual Nodes

Keys more evenly distributed and migration is evenly spread out.
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 ... 1/10000 ... 1/100000
Zipf “Heavy Tail” Distribution

\[ \frac{1}{k^\alpha} \]

Popularity

Rank
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: $O(\text{# vnodes})$
- $\text{table[hash(key)]->server}$
- Same table on every client
- Shard master adjusts table entries to balance load
- Periodically broadcast new table
Split hash range into buckets, assign each bucket to a server, busy server gets fewer buckets, can change over time
Split hash range into buckets, assign each bucket to a server, low load servers get more buckets, can change over time
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

hash(“despacito”) -> 1

1 -> Cache 1

2 -> Cache 2

Cache 1

Cache 2

Cache 3
Power of Two Choices

hash("despacito")

Cache 1
Cache 2
Cache 3
Power of Two Choices

hash("paxos")

2

3

Cache 1

Cache 2

Cache 3
Power of Two Choices

hash("paxos")

2

3

Cache 1

Cache 2

Cache 3
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
“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
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