Controlling Failure Through Sharding

Wes Chow / CTO, Chartbeat / @weschow
Avoiding Work

One way to improve performance is to not do work.

Cache the heavy things, like…?
Memcache

- In memory key-value store.
- Keys and values are opaque.
- Objects have TTLs (but we’ll ignore).
- High performance.
- Single server.
The Memcache Pattern

val = mc.get(key)  # key = ‘wes.followers’
if val is None:
    # lookup failed
    val = compute_value()  # db query
mc.put(key, val)

return val
Done!

Thanks for coming.

Questions?
Partitioning

In a multi-server environment:

- Load balancing: where should we do work?
- Distributed databases (sharding): where should data live?
Key Distribution Goals

● Faster than `compute_value()`.
● More reliable than target servers.
● Even distribution.
● Stateless (mostly) clients.
Sharded Memcache Pattern

```python
mc = mc_servers[hash(key) % len(mc_servers)]
val = mc.get(key)
if val is None:
    # lookup failed
    val = compute_value()
    mc.put(key, val)

return val
```
Hashmod, 3 nodes

rate = 1,000 requests / sec

1M keys

100% hit rate ~ 5 seconds
Hashmod, 3 + 1 nodes

rate = 1000 requests / sec

1M keys

add node at 5 seconds
hash(key) $\rightarrow$ 3 Nodes

mc = mc_servers[hash(key) % len(mc_servers)]

0 $\rightarrow$ 0
1 $\rightarrow$ 1
2 $\rightarrow$ 2
3 $\rightarrow$ 0
4 $\rightarrow$ 1
5 $\rightarrow$ 2
6 $\rightarrow$ 0
...

...
**hash(key) ➔ 3 & 4 Nodes**

\[
\begin{array}{ll}
0 & \rightarrow 0 \\
1 & \rightarrow 1 \\
2 & \rightarrow 2 \\
3 & \rightarrow 0 \\
4 & \rightarrow 1 \\
5 & \rightarrow 2 \\
6 & \rightarrow 0 \\
\cdots & \cdots
\end{array}
\]

\[
mc = mc\_servers[\text{hash(key)} \% \text{len(mc\_servers)}]
\]
hash(key) % 3 or 4
Observations

1. Hashmod only looks at # of nodes.
2. To minimize node churn, take nodes into account.

Let’s plot the nodes on a line.
Evenly Spaced Nodes

1. Convert nodes to number and put on line.
2. Convert key to number and put on line.
3. The node the key belongs to is the one immediately to the right.
1. Hash nodes to number and put on line.
2. Hash key to number and put on line.
3. The node the key belongs to is the one immediately to the right.
Law of Large Numbers

A principle of probability and statistics which states that as a sample size grows, its mean will get closer and closer to the average of the whole population.
Consistent Hashing

1. For each node, generate $M$ vnodes.
2. Hash every vnode and put it on a line.
3. To lookup a key, hash the key then binary search the line.
Hashed Nodes

$N = 3, M = 3$

#0:2 means “hash of node 0 vnode 2”
Adding Nodes

~ 1/N data gets shifted to new node 3. Intuition?
Consistent Hashing

rate = 1,000 requests / sec
add node at 5 seconds
use consistent hashing
Comparison

hashmod

consistent
Observations

- Hashmod only looks at # of nodes.
- To minimize node churn, take nodes into account.

Hashmod’s failure is the re-assignment of keys. Let’s keep key assignment fixed.
Node Sets

A B C (50% of A)
A C B (other 50% of A)
B A C
B C A
C A B
C B A

3! = 6 permutations
Failure of A

A B C
A C B
B A C
B C A
C A B
C B A

- B C
- C B
B - C
B C -
C - B
C B -
| D | A | B | C |   | A | D | B | C |   | A | B | D | C | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D | A | C | B |   | A | D | C | B |   | A | C | D | B | A | C | B | D |
| D | B | A | C |   | B | D | A | C |   | B | A | D | C | B | A | C | D |
| D | B | C | A |   | B | D | C | A |   | B | C | D | A | B | C | A | D |
| D | C | A | B |   | C | D | A | B |   | C | A | D | B | C | A | B | D |
| D | C | B | A |   | C | D | B | A |   | C | B | D | A | C | B | A | D |
Whereto, Node D?
1. Create (deterministic) ordered list of nodes for each key.
2. k belongs on first node in list.
3. Adding a new node explodes this ordered list into a new set of lists.
On key lookup:

1. For each node n, hash(k + n).
2. Sort hashes.
3. k belongs on node of first hash.
Rendezvous Example

k = “wes”

nodes = [“A”, “B”, “C”]

hashed nodes = [h(“wesA”), h(“wesB”), h(“wesC”)]
  = [2422(A), 1638(B), 9737(C)]

sorted = [1638(B), 2422(A), 9737(C)]

“wes” belongs on node B. What if we add D?
Rendezvous Hashing

rate = 1,000 requests / sec

add node at 5 seconds

use rendezvous hashing
Comparison

Consistent

Rendezvous
Consistent Hashing

- Lots of work at start.
- Typical M = 160.
- 1 hash calculation.
- Assignment / lookup is binary search.

\[ \# \text{ comparisons } \sim 7 + \lg_2(N) \]
Rendezvous Hashing

- No work at start.
- Lookup runs $N + 1$ hash calculations.
- $O(N)$ assignment / lookup.

But we can optimize with partitioning...
Poison Pills

- Expensive database / external queries.
- Buggy handler.
- Garbage input crashes service.
- Unhandled input (unicode) crashes service.
Random Selection

- 6 services
- 1,000 requests / sec
- 1% users trigger failure
- Service down for 1 sec
- 1 attempt
Random + Retry

- 6 services
- 1,000 requests / sec
- 1% users trigger failure
- Service down for 1 sec
- 3 attempts
Comparison
What Can We Do?

There's a big difference between mostly dead and all dead.

Mostly dead is slightly alive.
Retry sets:

1. 50% → A, B, C
2. 50% → D, E, F

if hash(key) < 0.5 then
    use set #1
else
    use set #2
Limit Blast Radius, \( N = 5 \)

1. 50% → A, B, C
2. 50% → D, E, C
Limit Blast Radius, N = 5

1. 50% → A, B, C
2. 50% → D, E, C
Two Retry Sets

- 6 services
- 1,000 requests / sec
- 1% users trigger failure
- Service down for 1 sec
- 3 attempts
- 2 retry sets
- Limited to ~50% damage
Observations

- Overlapping sets not a bad thing, assuming proper load distribution.
- The more sets, the smaller the blast radius.
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A B C</td>
<td>A B D</td>
<td>A B E</td>
<td>A B F</td>
<td>A C D</td>
<td>A C E</td>
</tr>
<tr>
<td>2</td>
<td>A C D</td>
<td>A C E</td>
<td>A C F</td>
<td>A D E</td>
<td>A D F</td>
<td>A E F</td>
</tr>
<tr>
<td>3</td>
<td>A D F</td>
<td>A E F</td>
<td>B C D</td>
<td>B C E</td>
<td>B C F</td>
<td>B D E</td>
</tr>
<tr>
<td>4</td>
<td>B C F</td>
<td>B D E</td>
<td>B D F</td>
<td>B E F</td>
<td>C D E</td>
<td>C D F</td>
</tr>
<tr>
<td>5</td>
<td>C D E</td>
<td>C D F</td>
<td>C E F</td>
<td>D E F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 6 Choose 3 + 3 Deaths

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>F</td>
<td>A</td>
<td>E</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>F</td>
<td>B</td>
<td>D</td>
<td>E</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>C</td>
</tr>
</tbody>
</table>
Shuffle

- 6 services
- 1,000 requests / sec
- 1% users trigger failure
- Service down for 1 sec
- 3 attempts
- Shuffle sharded
- Limited to ~5% damage
100% chance to lose 5% of data
The Fixed Cost of Failure

- Admitting failure on Twitter has high fixed cost. Failing for 50% of customers not much worse than 5%.
- Going to tape has high fixed cost. Restoring 1 TB not much worse than restoring 1 GB.
Apache Kafka

- High throughput message broker.
- Topics broken into K partitions.
- Each partition handled by primary/secondaries.
- Classic master/slave replication.
- Consumers subscribe to subset of partitions.
Chartbeat Pings

- Browser sends beacon to our servers.
- 275,000 / sec into “pings” topic.
- “pings” topic broken into 144 partitions.
- 6 brokers.
Sample Assignment

144 partitions, 6 nodes, R=2

pings:0 → {A B}
pings:1 → {C D}
pings:2 → {A F}
pings:3 → {C A}
...

Sample Failure

144 partitions, 6 nodes, R=2

pings:0 → {- B} (B elected primary)
pings:1 → {C D}
pings:2 → {- F} (F elected primary)
pings:3 → {C -} (no change)
...

Simple Assignment

\{A \ B\}
\{C \ D\}
\{E \ F\}

- \( \frac{1}{3} \) data on each set.
- Single node restore hard.
- If \( R=3 \), restores are faster, but…?
Cleverer Assignment

\{A, B\}
\{A, D\}
\{C, D\}
\{C, F\}
\{E, F\}
\{E, B\}

- \(\frac{1}{6}\) data on each set.
- Single node restores from 2 others.
- R=2 not 3 :)
Terminology

- $N =$ number of nodes
- $R =$ replication factor (# of copies of data)
- $S =$ scatter width
Simple Assignment

\{A \ B\}
\{C \ D\}
\{E \ F\}

N=6, R=2, S=1
Cleverer Assignment

\{A \ B\} \quad N=6, \ R=2, \ S=2
\{A \ D\}
\{C \ D\}
\{C \ F\}
\{E \ F\}
\{E \ B\}
Shuffle Sharding S?

A B C   A B D   A B E   A B F
A C D   A C E   A C F   A D E
A D F   A E F   B C D   B C E
B C F   B D E   B D F   B E F
C D E   C D F   C E F   D E F
Shuffle Sharding S?

A restores from all other nodes, S=N−1
Shuffle Sharding

A B C   A B D   A B E   A B F
A C D   A C E   A C F   A D E
A D F   A E F   B C D   B C E
B C F   B D E   B D F   B E F
C D E   C D F   C E F   D E F

100% chance to lose 5% of data
S = 4
$S = 4$

A B C
B C D  A restores from B, C, E, F
C D E
D E F  each set = 17% of data
E F A  $P_F = \frac{6}{6c3} = 30\%$
F A B
$S = 2$

A B C
D E F  A restores from B, C

each set = 50% of data

$P_F = \frac{2}{6c3} = 10\%$
Expected Data Loss

\[ S = N - 1 \]

\[ 100\% \times 5\% = 5\% \]

\[ S = 4 \]

\[ 30\% \times 17\% = 5\% \]

\[ S = 2 \]

\[ 10\% \times 50\% = 5\% \]
The Importance of $S$

- Affects $P_F$.
- Affects speed of restoring single node.
- Low $S = \text{low } P_F$, high damage, slow restore
- High $S = \text{high } P_F$, low damage, fast restore
Tiered Replication

To construct a copyset:

1. Order nodes from smallest to largest scatter width.
2. Pick first $R$ nodes.

Repeat until all nodes have $SW \geq S$. 
TR With Constraints

To construct a copyset:

1. Order nodes from smallest to largest scatter width.
2. Pick first $R$ nodes satisfying constraints.

Repeat until all nodes have $SW \geq S$. 
Shuffle sharding confuses service level checks.

Shuffling requires knowledge of poison dimension.

Copysets is NP-Hard in general.

Tradeoffs. Embrace or reduce catastrophe?
Whew

- Basic sharding easy. Failure scenarios fraught.
- Straightforward implementations, but hard to internalize.
- Get it right once. (ketama, trepl)
- Finally, Murphy++.
Now We’re Really Done.

Consistent Hashing: Karger et al. 1997

Rendezvous Hashing: Thaler & Ravishankar 1996

Shuffle Sharding: ?

Tiered Replication: Cidon et al. 2015