Initial Problem

Hot spots:

- Occurs any time when large number of clients wish to access the data from single server

- Multiple requests causes swamping, rendering it useless.

Terminologies

c: Set of caches

C: Number of caches

p: Set of all pages

δ(m1,m2): Latency - time taken for a message from machine m1 to arrive at m2

q: Number of requests a cache must see before bothered to store a copy of page

Random Trees

- Simplifications to the model:

1. All machines know about all caches
2. $\delta(mi,mj) = 1$ for all $i \neq j$
3. All requests are made at same time

- Static Model - Only one batch of requests
- Basic idea:
  - extension of “tree of caches” - use a different, randomly generated tree for each page
  - ensures that no machine is near the root for many pages- load balancing

Random Trees

<u>Protocol :</u>

- Associate a rooted d-ary tree (abstract tree) with each page
- Request form a 4-tuple: identity of the requester, name of desired page, sequence of nodes, sequence of caches
- Mapping :
  - root always mapped to server for page
  - other nodes are mapped to caches by hash function:

$$h: p \times [1\ldots\ldots C] \rightarrow c \text{ - distributed to all browsers and caches}$$

- Parameter q avoid copies of pages with fewer requests

Random Trees

- Browser (wants a page)
  - picks a random leaf to root path
  - maps nodes to caches with h
  - asks the leaf node for the page
  - request - name of browser, name of page, path, result of mapping
- Cache (receives a request)
  - first checks if it is caching a copy of page or is in the process of getting one to cache
  - if so, it returns the page to the requester
  - otherwise, it increments the counter for page and node it is acting as and asks the next cache on path
  - if counter reaches q - caches the copy of page
- Server (receives a request)
  - sends the requester a copy of page

Random Trees

Latency:

- If request is forwarded from leaf to root
  - latency = $2 \log_d C$
- Latency is less if the request is satisfied with a cached copy
- In practice, time required to obtain large page is not multiplied by the number of steps in path it travels

Random Trees

Swamping :

- The following theorem provides high probability bounds on the number of requests a cache gets.

if h *is chosen uniformly and at random from the space of functions* p X [1….C] → c, *then with probability at least* 1-1/N, *where* N *is a parameter, the number of requests a given cache gets is no more than*

$$\rho(2\log_d C + O(\log N/\log\log N)) + O(dq\log N/(\log(dq/\rho * \log N)) + \log N)$$

Random Trees

Proof:

1. Request to leaf nodes

- Each request coming for page p goes to a random leaf node in that page's tree
- L denotes the no. of leaf nodes in each abstract tree
- Associate a weight $w_p=r_p/L$ with each abstract leaf node of p's tree
- Machine m has 1/C chance of being at an arbitrary leaf node
- Let V_{pj}= event when j^{th} leaf node is assigned to m
- $V_{pj}=1$, with probability 1/C and $V_{pj}=0$ otherwise
- Bounding the total weight $W=\Sigma w_p V_{pj}$ assigned to m

Random Trees

- We would use Chernoff Bound here.
- W is weighted sum of poisson variables with weights possibly greater than 1
- But, each weight $w_p = r_p / L <= \rho / d(d-1)$
- So we can apply chernoff bounds to $W/\rho/d(d-1)$
- This gives bound of $O(\rho \log N / \log \log N)$ on W which holds with probability 1-1/N --------------1

2. Request to internal nodes

- No internal node gets more than dq requests
- Since only R requests in all, bound the no. of abstract nodes that get dq requests

Random Trees

- In fact bound, n_j , the no. of nodes over all trees that receive between 2^j and 2^{j+1} requests where 0<= qj <= log(dq)-1
- $\Sigma r_p <= R$
- Since, each of the R requests gives at most $\log_d c$ requests, total no. of requests is no more than $R\log_d C$
- So, $\Sigma_{j=0}^{\log(dq)-1} 2^j n_j <= R\log_d C$

Lemma: The total no. of internal nodes that receive at least qx requests is at most 2R/x, x>1

Proof:

- Any node gets at most q requests from each child, a node that gets at least qx

Random trees

requests must have downward degree of at least $x>1$.

- Consider all nodes u with downward degree one
- Let v and w be parent and child of u respectively
- Replace all such nodes with degree 1 by a single edge from v to w.
- Since $x>1$, we now have a tree where each node has downward degree of at least 2
- The no. of leaves in tree is no more than no. of requests, r_p
- So, if there are y nodes with downward degrees of at least x, the $xy<=2r_p$ and so $y<=2r_p/x$
- Thus, total no. of nodes over all trees which receive at least qx requests is no more than $\Sigma 2r_p/x=2R/x$

Random Trees

- For x=1, there can clearly be no more than $R\log_d C$ requests
- The lemma showed: n_j is at most $2R/2^j$, but for j=0, n_j is at most $R\log_d C$
- Assignments of nodes are independent, the prob that a machine m gets more than z of these nodes is at most $({}^{n_j}C_z)\ (1/C)^z\ (en_j/C\ z)^{\ z}$
- In order for RHS to be as small as 1/N, we set

$$z=\Omega(n_j/C+\log N/\log((C/n_j)\log N))$$

- Latter will be present only if $C/n_j\log N>2$, so z is $O(n_j/C+\log N/\log((C/n_j)\log N))$ with prob at least 1-1/N.
- So, with prob at least 1-log(dq)/N, the total no. of requests received by m due to internal nodes is of order of

Random Trees

- $\Sigma_{j=0}^{\log(dq)-1} 2^{j+1}(n_j/C+\log N/\log((C/n_j)\log N))$

= $2\,\rho\log_d C + O(dq\log N/\log((dq/\rho)\log N))+\log N)$ ---------- 2

hence from 1 and 2

$\rho\ (2\log_d C + O(\log N/\log\log N))+O(dq\log N/\log((dq/\rho\ )\log N)+\log N)$

Consistent Hashing

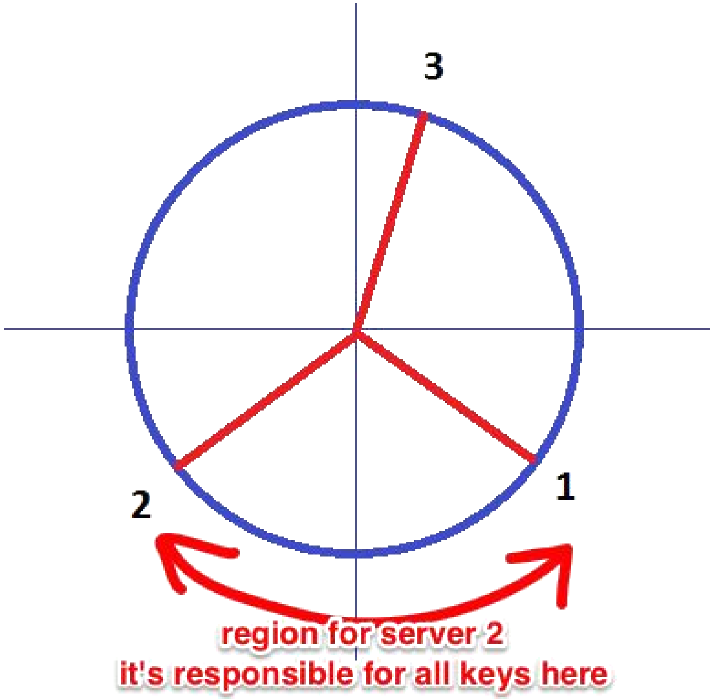
Problem: Evenly distributing objects across caches such that each client should know where to query for a particular object.

Typical Solution: The server can use a hash function to evenly distribute objects across caches.

But,

- When a new cache is added or removed, every item would be hashed to a different location.

Consistent Hashing



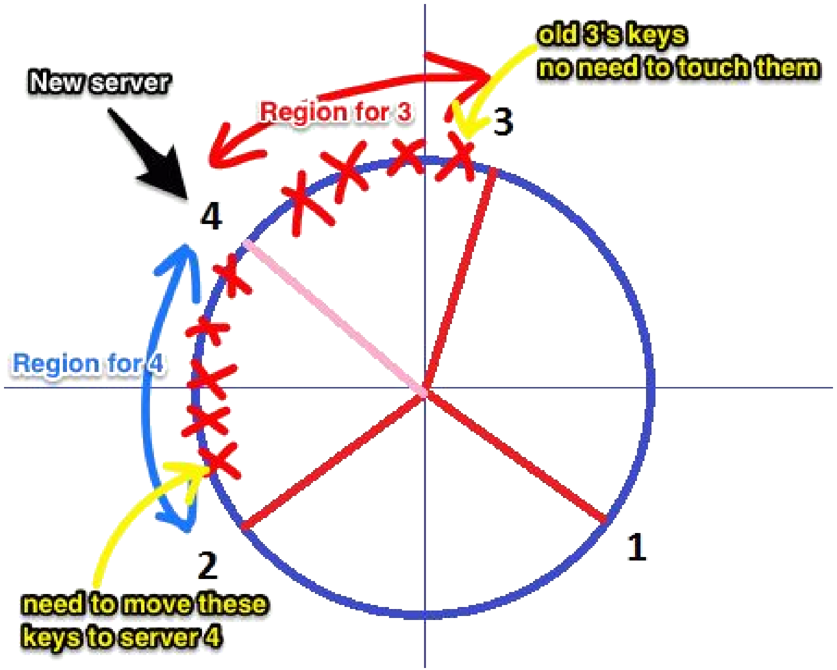
A particular client needs to only aware of a set of caches.

- In *consistent hashing* a hash function is viewed as a ring
- Each server has its own key region (its "position" on the ring)

For example

- The key region for 2nd server is the area between 1 and 2
- Only 2 is responsible for keys in that region.

Consistent Hashing



Suppose we want to add a new server

- we just pick some area and divide it into 2 parts
- Assign the new server one of these two.
- The keys that happen to be in that region are moved to the new server

- Each server knows the key range which it manages
- We can route the request to the server that is close to the key we're looking for

Consistent Hashing

Virtual Nodes

There are some challenges with this basic approach

- random position assignment may lead to non-uniform data/load distribution
- heterogeneity in performance is assumed (that is, we assume that all the servers have same performance)

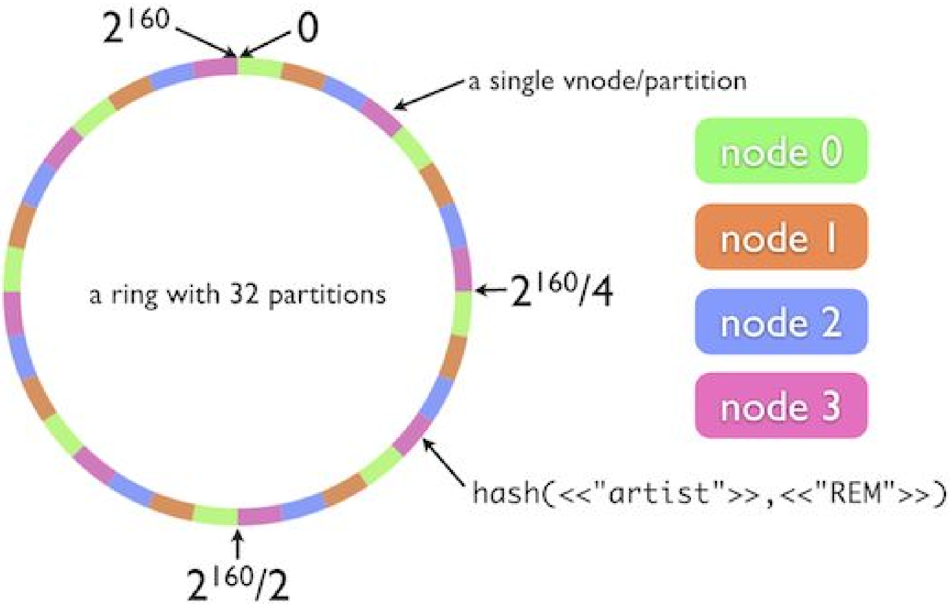
A variant of Consistent Hashing algorithm addresses this issue:

- instead of mapping a single node to the ring, each node gets multiple points there.(*virtual nodes*)

A virtual node looks like a single node, but it refers to the real node.

Advantages: Load balancing.

Consistent Hashing



Properties of consistent hashing.

- "spread"
- "load"
- "smoothness"
- "balance"
- "monotonicity"

- Ranged hash function: $f : 2^{\beta} \times I \rightarrow \beta$
- View: The set of caches of which a particular client is aware. $V \subseteq \beta$
- $f(v, i) = f_v(i)$ = the bucket to which item i is assigned in view v.

Consistent Hashing

Balance: Distributing items among buckets in a balanced fashion.

A ranged hash family is balanced if for a given view (V), a set of items, and a randomly chosen hash function, with high probability, the fraction of items mapped to each bucket is O(1/|V|)

Monotonicity: Item may move from an old bucket to a new bucket , but not from one old bucket to another. (intuition about consistency)

A ranged hash function f is monotone if for all views $V_1 \subseteq V_2 \subseteq \beta$,

$fv_2(i) \in V_1$ implies $fv_1(i) = fv_2(i)$.

Consistent Hashing

Spread: Let $V_1 ... V_v$ be the set of views, altogether containing C distinct buckets and each individually containing at least C/t buckets. The spread σ(i) is the quantity $|\{fv_j (i)\}^V_{j=1}|$. The spread of a hash function is the maximum spread of an item. When each person tries to assign an item i to a bucket, there are at most σ(i) different opinions about which bucket should contain this item. Good consistent hashing function have low spread.

Load: For a ranged hash function *f* and bucket *b*, the load λ(*b*) is the quantity

$|\cup_v f_v^{-1}(b)|$. The load of a hash function is the maximum load of a bucket. There are at most λ(*b*) distinct items that at least one person thinks belongs to the bucket.