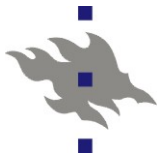
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Overlay and P2P Networks

Consistent Hashing

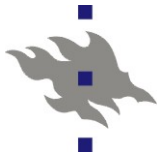
Prof. Sasu Tarkoma

31.1.2013



Contents

- Structured networks
- Foundations
- Cluster-based structures



Structured Overlays

Structured overlays are typically based on the notion of a **semantic free index**

They utilize **hashing** extensively to map data to servers

The **cluster**-based techniques typically can guarantee a very small number of hops to reach a given destination

The decentralized **DHTs** balance hop count with the size of the routing tables, network diameter, and the ability to cope with changes

Consistent hashing

Consistent hashing was first introduced in 1997 as a solution for distributing requests to a dynamic set of web servers

In this solution, incoming messages with keys were mapped to web servers that can handle the request

Consistent hashing has had dramatic impact on overlay algorithms

DHTs utilize consistent hashing to partition an identifier space over a distributed set of nodes. The key goal is to keep the number of elements that need to be moved at minimum

Consistent hashing continued

In most traditional hash tables a change in the number of array elements causes nearly **all keys** to be remapped

They are therefore useful for balancing load to a fixed collection of servers, but not suitable for dynamic server collections

Consistent hashing is a technique that provides hash table functionality in such a way that the addition or removal of an element does **not significantly change** the mapping of keys to elements

The technique requires only *K/n* keys to be remapped on average, where *K* is the number of keys, and *n* is the number of nodes

Ranged hash functions

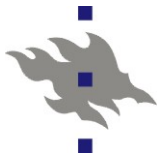
Hashing applied to the distributed case

Ranged hash functions are hash functions that depend on the set of available buckets

A typical ranged hash function hashes items to positions in some space

Then assigns each item to the nearest available bucket

As the set of buckets changes, an item may move to a new nearest available bucket



Another view

A ranged hash function changes minimally as the range of the function changes

Range changes when a server is added or removed

Ranged hash with a ring

Items and buckets are mapped to a uniformly random place on continuous unit ring [0,1).

Each item is assigned to the closest possible bucket.

Bucket order determines placement on the ring.

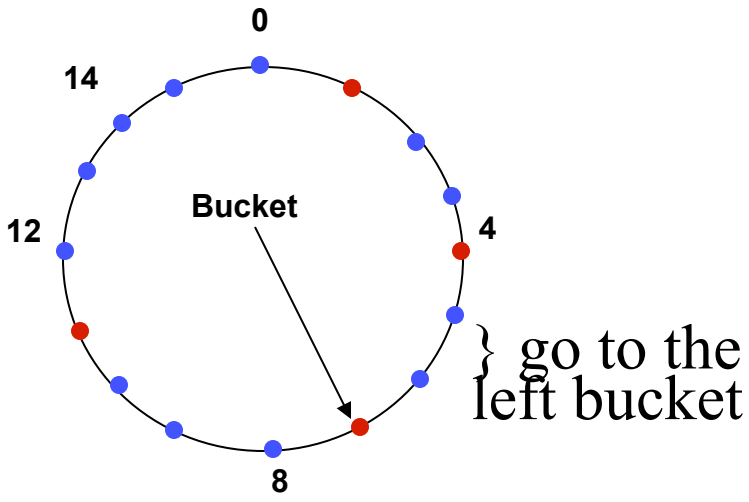
Optimality proven for growth-restricted metric spaces

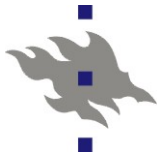
Given point q and distance d, the number of points within distance 2d is at most constant factor larger than within distance d.

J. Aspnes at al. Ranged Hash Functions and the Price of Churn. SODA 2008.

Example of Consistent Hashing

- Creating the structure
  - Assign each of C hash buckets to random points on mod 2^n circle, where, hash key size = *n*
  - Map object to random position on circle
  - Hash of object = closest clockwise bucket





Properties of ranged hash functions

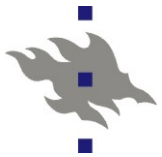
Monotone

Each item has its own preference list and hashes to the first available bucket

This minimizes rearrangement cost

Multiple virtual locations for the item are possible

Can be described with a preference matrix
Items and the buckets



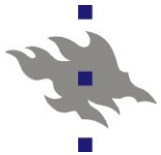
Properties of Consistent Hashing I

A **view** is a subset of the buckets (cache servers available from certain part of the network)

Consistent hashing uses a **ranged hash function** to specify an assignment of items to buckets for every possible view

A ranged hash family is said to be **balanced** if given a particular view, a set of elements, and a randomly chosen function from the hash family, with high probability the fraction of items mapped to each bucket is $O(1/|V|)$, where V is the view

In other words, items are uniformly distributed over the buckets of the view

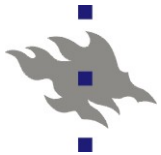


Properties of consistent hashing II

Load: A balanced ranged hash function distributes load **evenly** across the buckets

Monotonicity says that some items can be moved to a new bucket from old buckets, but not between old buckets. The aim is to preserve an even distribution

Spread is about ensuring that at least a constant fraction of the buckets are visible to clients



Problem

Having only one location for a bucket is not good

Does not ensure good spread

Solution: have multiple virtual locations for a bucket

Implication: when removing / adding a bucket, have to move data from several servers

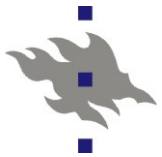
Replication with virtual buckets

One point is not sufficient to characterize a bucket due to the required properties.

A bucket is replicated $\kappa \log(C)$ times, where C is the number of distinct buckets, and κ is a constant

The log(C) term comes from the theory, basically it is needed to get the good fraction $O(1/|V|)$ of buckets to servers

When a new bucket is added, only those items are moved which are closest to one of its points. Similarly for the removal of a bucket.



Example of a ranged hash function (RHF)

Let I be the items, C the caches, and V the views. V_i is a subset of C.

RHF is a map that takes a view (all possible views $2^{C)}$ and hashes it to a cache in which the item can be found:
$h: 2^C \times I \rightarrow C$

For an item: pick a point r uniformly and independently at random

For the buckets: pick a set of $\kappa \log C$ points uniformly and independently at random.

For an item (V_i,i) map it to the first bucket b in V_i that is encountered clockwise starting from r.

Bad examples

Pick b in V at random: bad spread properties (the preference list of many buckets is needed, nearest clockwise bucket chosen)

Take mod of the number of caches in a view: good balance but not smooth (e.g. problems when adding or removing a server)

```
public class ConsistentHash<T> {
  private final HashFunction hashFunction;
  private final int numberOfReplicas;
  private final SortedMap<Integer, T> circle =
    new TreeMap<Integer, T>();

  public ConsistentHash(HashFunction hashFunction,
    int numberOfReplicas, Collection<T> nodes) {
    this.hashFunction = hashFunction;
    this.numberOfReplicas = numberOfReplicas;

    for (T node : nodes) {
      add(node);
    }
  }
  public void add(T node) {
    for (int i = 0; i < numberOfReplicas; i++) {
      circle.put(hashFunction.hash(node.toString() + ":" + i),
        node);
    }
  }
  public void remove(T node) {
    for (int i = 0; i < numberOfReplicas; i++) {
      circle.remove(hashFunction.hash(node.toString() + ":"+ i));
    }
  }
  public T get(Object key) {
    if (circle.isEmpty()) {
      return null;
    }
    int hash = hashFunction.hash(key);
    if (!circle.containsKey(hash)) {
      SortedMap<Integer, T> tailMap =
        circle.tailMap(hash);
      hash = tailMap.isEmpty() ?
             circle.firstKey() : tailMap.firstKey();
    }
    return circle.get(hash);
  } }
```

This code does not move data between buckets!
Should be added here

http://www.lexemetech.com/2007/11/consistent-hashing.html

Wraps around the circle here

Main point in consistent hashing

The technique requires only *K/n* keys to be remapped on average, where *K* is the number of keys, and *n* is the number of nodes

Used in most DHT algorithms

Developed by Karger et al. at MIT

Somewhat involved for example in Chord

Used by CDNs and caches

- Akamai