Consistent Hashing

SAD

P2P: Overlay Networks

- P2P applications need to
  - Track identities & IP addresses of peers
    - May be many and may have significant churn
  - Route messages among peers
    - If you don't keep track of all peers, this is "multi-hop"
- Overlay network
  - Peers doing both naming and routing
  - IP network becomes the low level transport

Structured Overlays

- Consider problem of data partition:
  - Have D resources
  - Need to store them in K servers
  - Need to distribute the load among the servers
  - Given document U, choose one of k servers to use to access it

Consistent Hashing

- Initial motivation (1997)
  - Web Page Caching
    - Increase scalability of web sites
    - Avoid swamping the "home" server for the page
    - Caching is good … but... Cache servers must be found
      - And we must avoid swamping them too
  - E.g. If we want to go to upv.es...
    - Where do we actually go?
    - How about the browser itself?
    - If stale...
      - Why not a shared cache?
        - Difficult

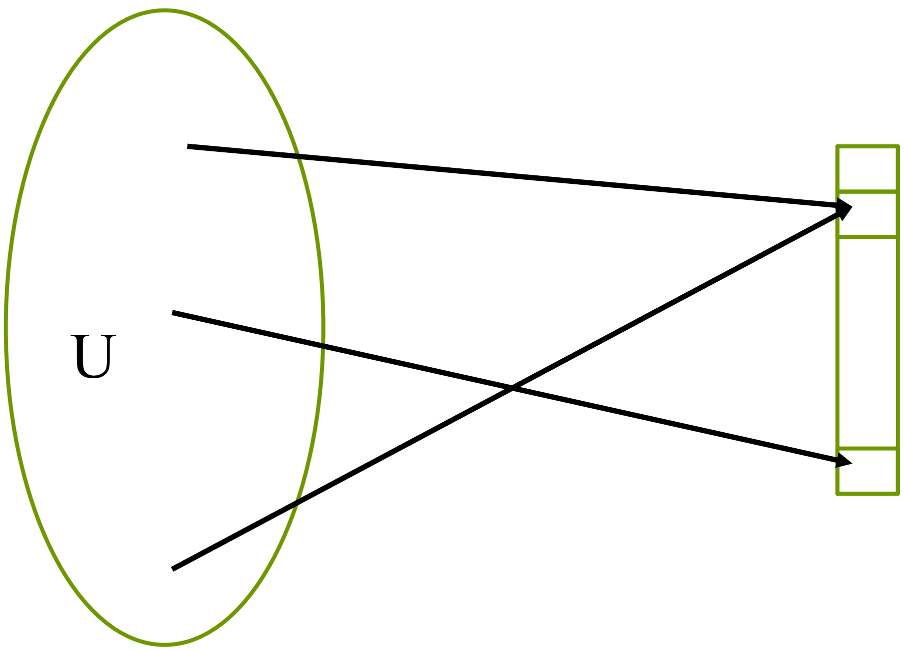
Consistent Hashing

- Benefits of shared Caching
  - Larger cache size
  - Higher chance of cache hit
  - Lower load on Servers
  - Faster Lookup
- BTW: Heard of Akamai? Anyone?
  - (12BUSD)
- Why is it difficult?
  - Cannot use just one machine
  - Spread over many different machines

Consistent Hashing

- OK, assume you have 100 cache servers
- Approaches
  - Poll each machine in the set and ask
    - ... Expected load per server?
    - … Is this reasonable?
  - Can we find a cache server number where to go given a URL?
    - Actually we can set up such a scheme
    - How about using a hash function?
    - Hash(url) → Server Number where to look for the url resource

Hash



Consistent Hashing

- Suppose we use modulo hashing
- Number servers 1..k
  - X = toNumber(url) (e.g. integer as a concat of bytes)
  - Place document with id X on server *i = (X mod k)*
    - *Thus hash(url) = modk (toNumber(url))*
- Problem?
  - Data may not be uniformly distributed
- MOD does not have distribution guarantees
  - Distribution governed by ID space distribution
    - Which may be very non-uniform
- THUS, potential overload problems on some of the servers
  - Poor balancing

Consistent Hashing

- A good hash function ideally must
  - Be easy to evaluate
  - Behave like a totally random function
    - Spread out universe U uniformly
- Difficult to design Good hashes
  - Better use already existing ones
  - Crypto hashes have good properties
    - MD5, SHA-1, …
- So, assuming we have a good hash function...
  - Place url on server *i = hash (url) mod k*
    - *(Why do we need the mod?)*

Consistent Hashing

- Are we done?
- Assume we add a new server
  - Now we have 101 servers
- Is hash(u) mod 100 == hash(u) mod 101?

Consistent Hashing

- Are we done?
- Assume we add a new server
  - Now we have 101 servers
- Is hash(u) mod 100 == hash(u) mod 101?
  - UNLIKELY
- THUS
  - We will need to relocate
  - CHANGING K FORCES resource relocation!
    - Rehash is an expensive operation
    - Disater for applications where K changes often
      - As is the case for web page caching
        - Original motivation
      - And is the case for P2P networks

Consistent Hashing

- Why k changes?
  - Failures occur
    - Thus the set of servers temporarily decreases
  - Load changes
    - It also increases to cope with the load
  - Thus → The set of servers changes over time
    - Guaranteed!
- How many do we need to relocate in the general case?

Consistent Hashing

- Why k changes?
  - Failures occur
    - Thus the set of servers temporarily decreases
  - Load changes
    - It also increases to cope with the load
  - Thus → The set of servers changes over time
    - Guaranteed!
- How many do we need to relocate in the general case?
    - Even with just 1 change, n/(n+1)*U resources will need to move
      - Seems impractical for large sets

Consistent Hashing

- On top of that
  - Reconfiguration takes time
  - Thus → Clients may have different views of the available set of servers
- Inconsistent view of who holds what
  - Need to maintain old state while moving
  - Need to detect when old state can be deleted
- Can we do better?
  - Hashing a good idea but…
  - We need something better
    - Enter Consistent Hashing

Consistent Hashing

- What do we want?
  - To have our cake and eat it !!!
- To Have a Hash-like functionality
  - Including its attractive programming interface
- To keep most resources assigned where they were before changes
  - Thus
    - Avoid downtime on server failures
    - Avoid overloading new servers on addition to the network
    - Avoid rehashing more keys than necessary on membership changes
- Can we do it?

Consistent Hashing

- Key Idea:
  - In addition to hashing the names of the resources…

Consistent Hashing

- Key Idea:
  - In addition to hashing the names of the resources…
- WE ALSO HASH THE Ids of the Servers!
  - To the same range as resources
  - They become also resources of sorts

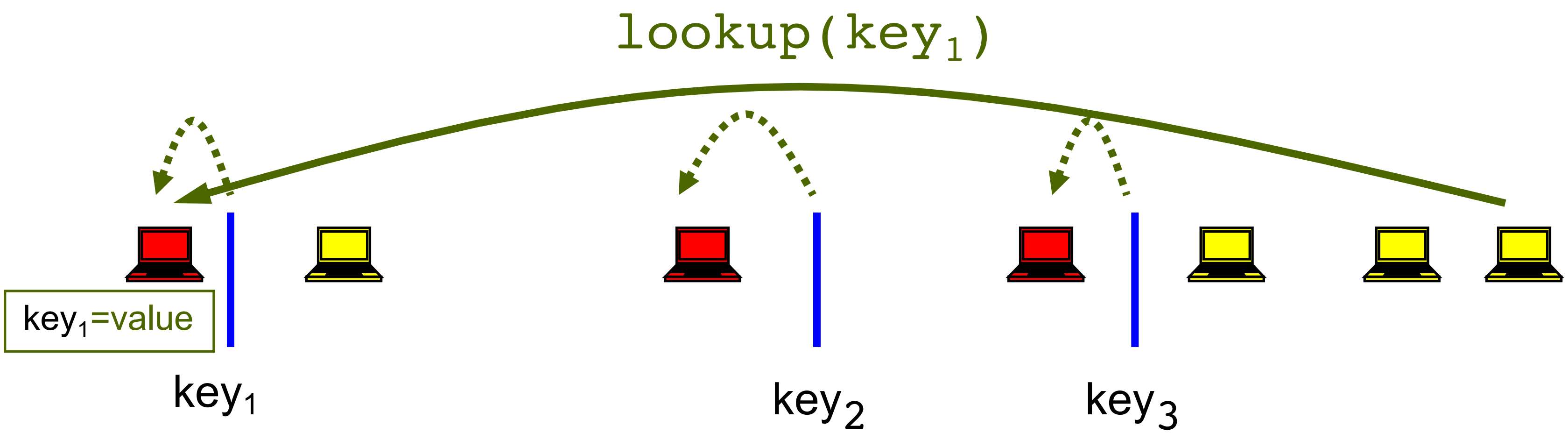
Consistent Hashing

- Key Idea:
  - In addition to hashing the names of the resources…
- WE ALSO HASH THE Ids of the Servers!
  - To the same range as resources
  - They become also resources of sorts

So WHAT?

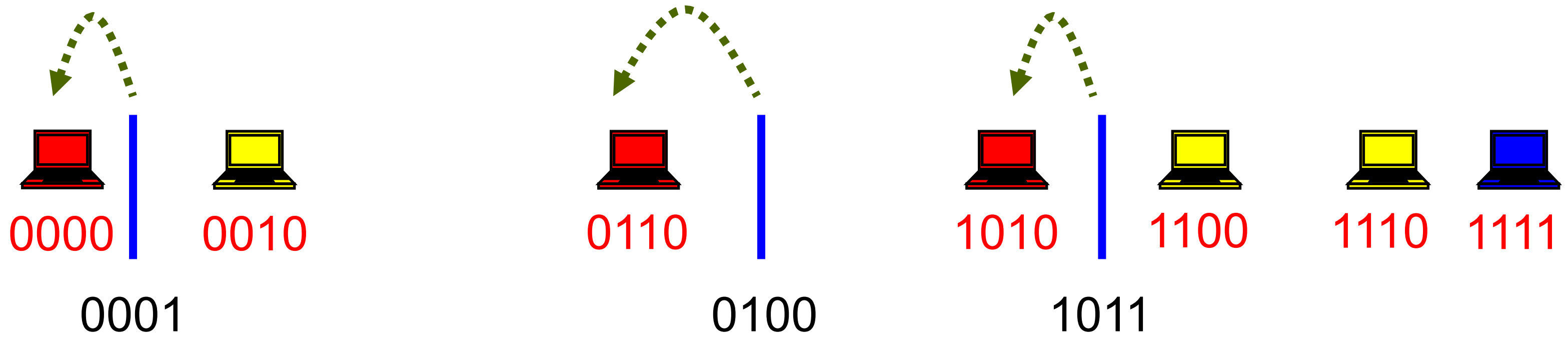
What is the big deal?

Consistent Hashing



- Consistent hashing partitions resource key-space among "nodes"
- Contact appropriate server to lookup/store key
  - Blue nodes determines red node is responsible for key_1
  - Blue node sends lookup or insert to red node

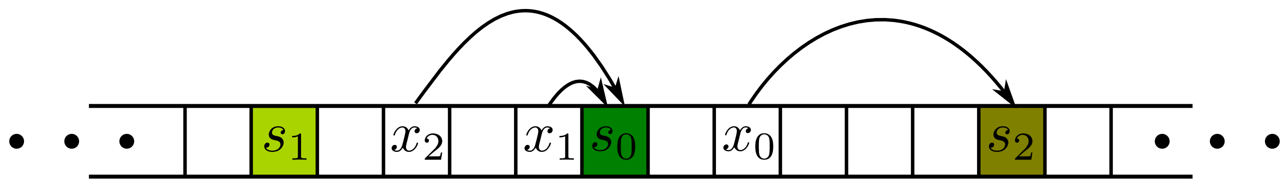
Consistent Hashing: Common Virtual Addresses



- Partitioning key-space among nodes
  - Nodes choose random identifiers: e.g., hash(IP)
  - Keys randomly distributed in ID-space: e.g., hash(URL)
  - Keys assigned to node "nearest" in ID-space
  - Spreads ownership of keys "evenly" across nodes

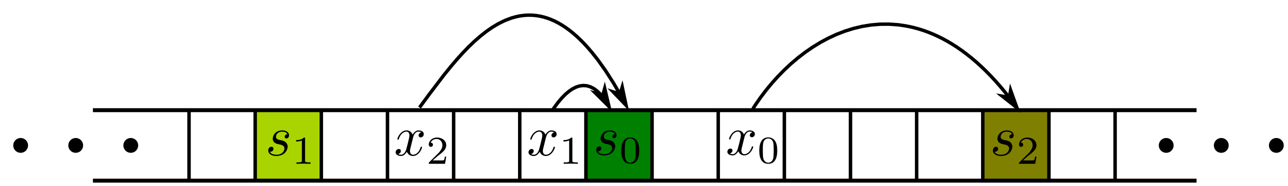
Consistent Hashing

- Each element of the array is a hash table bucket
  - Servers fall on particular buckets
  - Each resource, x_i is assigned to the server nearest to its right



Consistent Hashing

- Imagine bucket array
  - Each element of the array is a hash table bucket
  - Might be very large: e.g. 2^{32} for a 32 bit key length
    - Virtual, not implemented like that
  - Servers fall on particular buckets
  - Each resource, x_i is assigned to the server nearest to its right



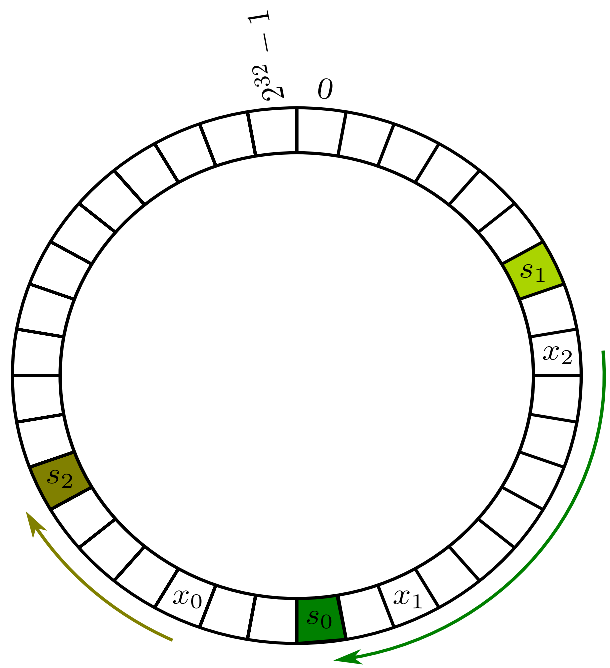
- But what do we do with the last key?

$$2^{32} - 1$$

- What is to its right?

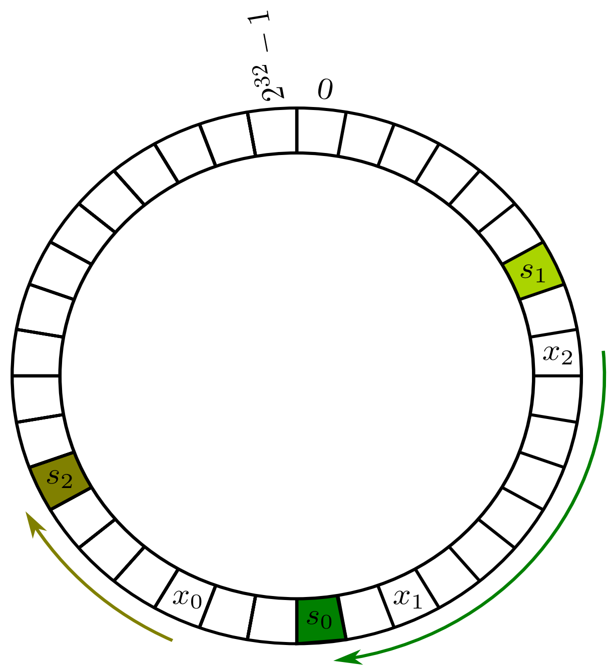
Consistent Hashing

- We can glue 0 and $2^{32}-1$ together forming a ring
  - Servers and resources map to bucket in the ring
  - Resources are assigned to the server closest clockwise
  - Solves the problem of the last object to the right



Consistent Hashing

- N servers
  - Partition the ring into N segments
  - Each server is responsible for one of those segments

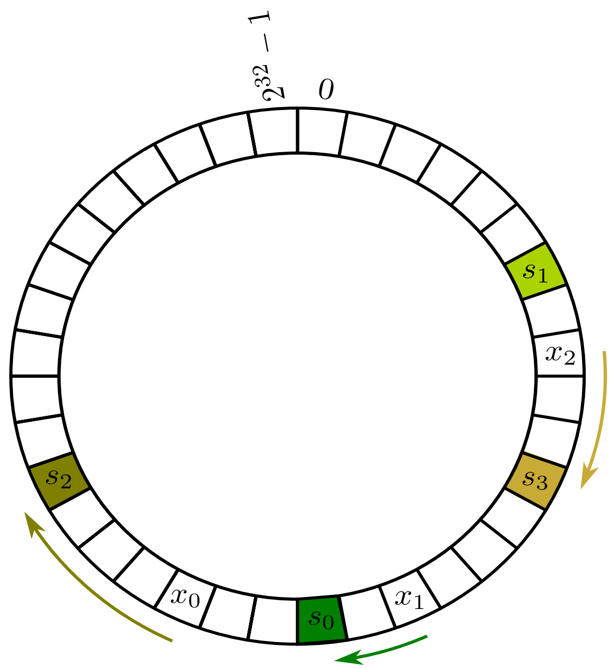


Consistent Hashing

- Nice properties
  - Assuming reasonable hash function
- By Symmetry
  - The expected load on each of the n servers is 1/n fraction of the resources
    - However there is some variance we need to reduce
  - If we add a new server we only need to move those resources that need to be stored there
  - …

Consistent Hashing

- If we add a new server, s_3
  - x_2 moves from s_0 to s_3
  - The rest remain at the same servers



Consistent Hashing

- Nice properties
  - Assuming reasonable hash function
- By Symmetry
  - The expected load on each of the n servers is 1/n fraction of the resources
    - However there is some variance we need to reduce
  - If we add a new server we only need to move those resources that need to be stored there
- Combined
  - We only need to move 1/n th of the resources
  - Compare to the naïve solution
    - Only 1/n th of the servers DO NOT HAVE to move

All on average…

Consistent Hashing

- So, how do we implement all this efficiently?
  - **Lookup** and **Insert** operations
  - Need efficient clockwise scan operation
    - For any x
    - Search for server s minimizing $\mathrm{h}(\mathrm{s})$, such that $\mathrm{h}(\mathrm{s}) \geq h(x)$
- Need data structure
  - Keys: $\mathrm{h}(\mathrm{s})$
  - Values: s
  - Fast **Successor** operation
- What can we use?

Consistent Hashing

- Hash table?

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...
- Heap?

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...
- Heap?
  - NO… Only keeps a partial order
    - Just to identify minimum
- Then what?

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...
- Heap?
  - NO… Only keeps a partial order
    - Just to identify minimum
- Then what?
- How about a binary search tree?
  - Implements total order
  - Provides a Successor operation

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...
- Heap?
  - NO… Only keeps a partial order
    - Just to identify minimum
- Then what?
- How about a binary search tree?
  - Implements total order
  - Provides a Successor operation
    - O(height of the tree)

Consistent Hashing

- Hash table?
  - NO… Does not keep order at all...
- Heap?
  - NO… Only keeps a partial order
    - Just to identify minimum
- Then what?
- How about a binary search tree?
  - Implements total order
  - Provides a Successor operation
    - O(height of the tree)
    - Height = $\log(n)$ when tree is balanced

Consistent Hashing

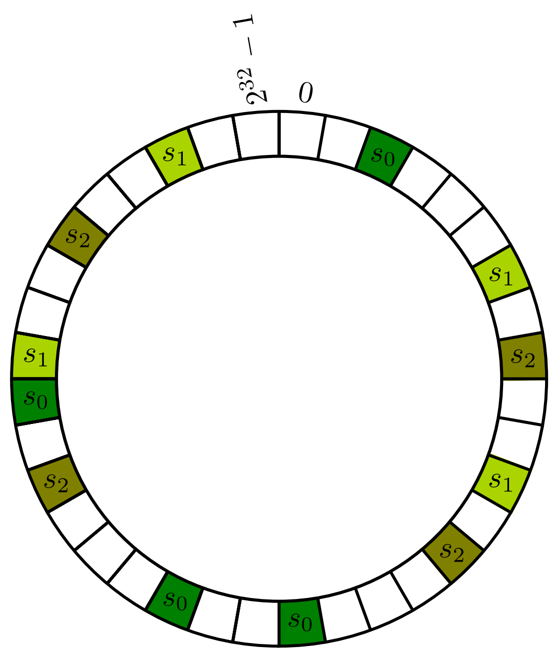
- Can we reduce the variance?
  - We expect 1/n th of the resources on each server on average
  - But it is very unlikely each server has that portion
    - Imagine getting a perfect partition of the ring: unlikely
- We can decrease variance by

Consistent Hashing

- Can we reduce the variance?
  - We expect 1/n th of the resources on each server on average
  - But it is very unlikely each server has that portion
    - Imagine getting a perfect partition of the ring: unlikely
- We can decrease variance by…
  - Making v virtual copies of each actual server
    - We can hash each one of them like this: hash(i,j), where
      - i is the identity of the server
      - j in 1..v is the identity of the j-th virtual server associated to server i.

Consistent Hashing

- Suppose v = 4, for servers {1,2,3}
  - We would have 4 x 3 = 12 buckets in the ring, 4 per server
  - When we search for x, we do as before hash(x)
    - Find the first bucket clockwise
    - Chose the server it belongs to



Consistent Hashing

- Again, by symmetry
  - Each server is expected to get 1/n th of resources on average
- But, the replication increases the number of stored keys by a factor of v
- But load variance across servers is reduced significantly

Consistent Hashing

- Again, by symmetry
  - Each server is expected to get 1/n th of resources on average
- But, the replication increases the number of stored keys by a factor of v
- But load variance across servers is reduced significantly
  - Some copies of a server will get more than 1/vn
  - But other will get less than 1/vn
  - Thus
    - We get the variance averaged out among the various copies of a server!!
- What is a good value for v?
  - Around $\log_2 n$ is large enough for variance
    - And small enough to avoid blowing up the size of the tree

Consistent Hashing

- Additional Properties of virtualizing Servers
  - Can be brought on-line gradually
    - Start with one virtual server
      - Gradually increase the number of virtual servers it holds
  - Can have a number of virtual servers adapted to its capacity
    - The ”larger” the server the more virtual servers it should have.
      - So it can take additional load
  - On failure/quitting
    - The load of the gone server is …