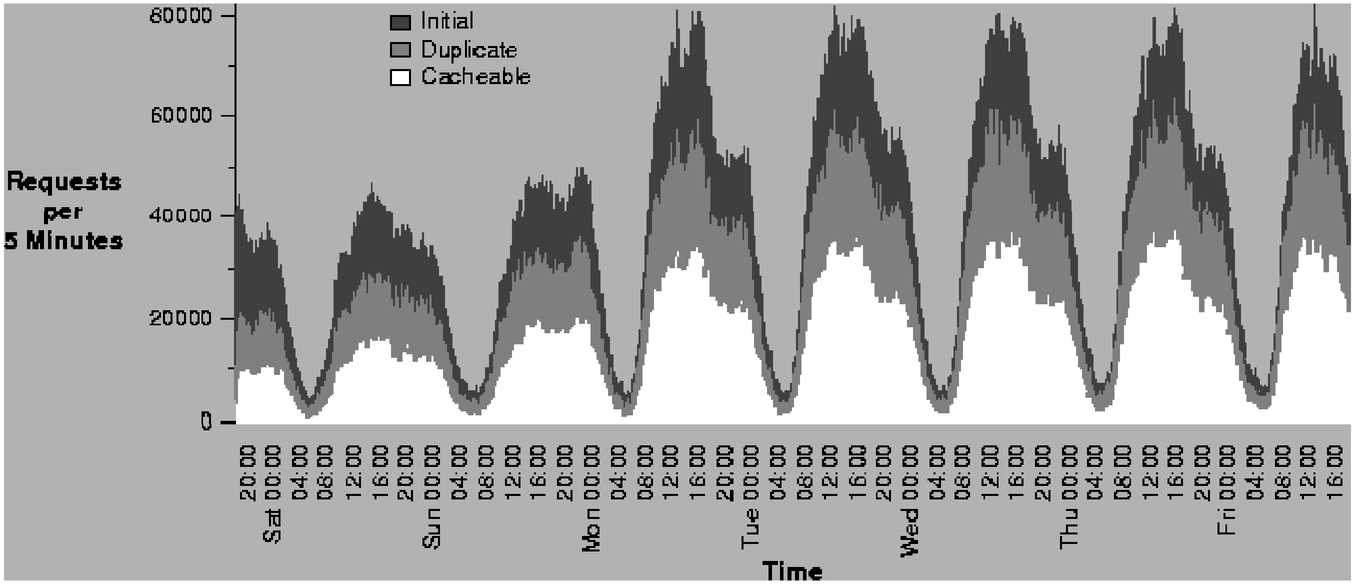
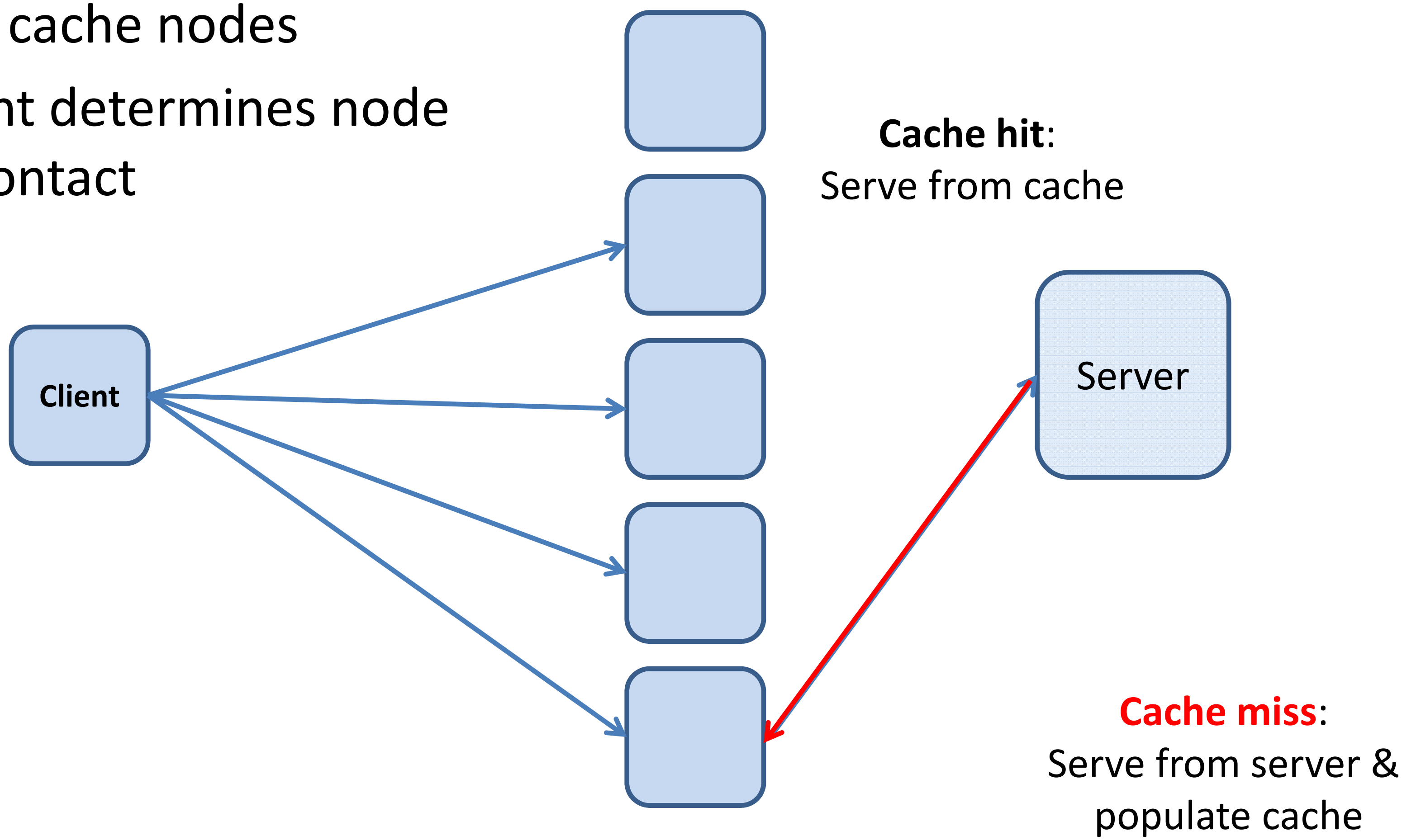
Consistent Hashing

Organization-Based Analysis of Web-Object Sharing and Caching
Alec Wolma et al.



Caching

- Five cache nodes
- Client determines node to contact



Problem: Mapping objects to caches

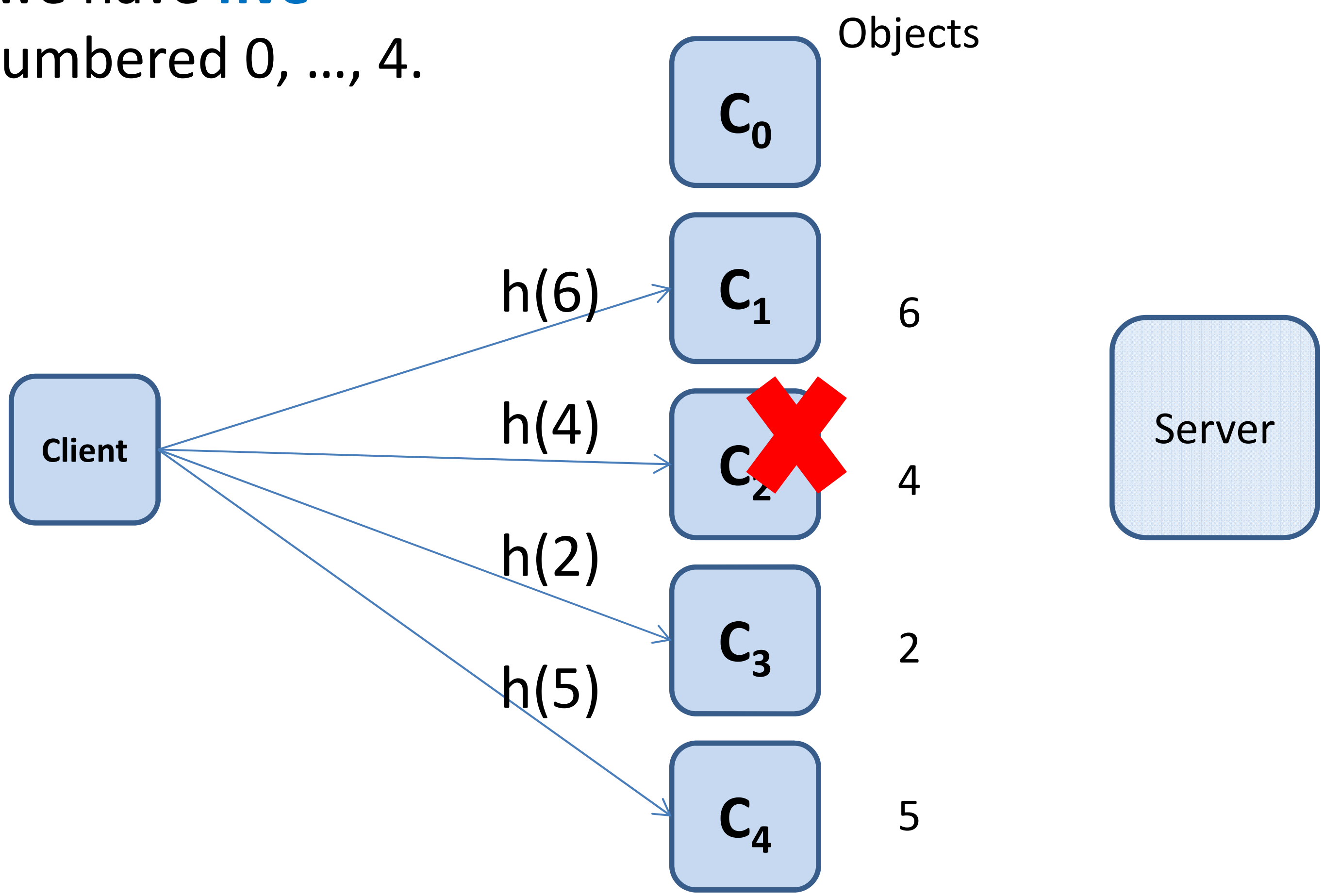
- Given a number of caches (e.g., cooperative caching, CDNs, etc.)
- Each cache should carry an **equal share of objects**
- **Clients** need to **know what cache to query** for a given object
- **Horizontally partition** (shard) object ID space
  - **Doesn't work** with **skewed distributions**: e.g., 10 servers, each handles 100 IDs, but all objects have IDs between 1-100 or 900-1000
- **Caches** should be able to **come and go** without disrupting the whole operation (i.e., non-effected caches)

Solution attempt: Use hashing

- **Map object ID** (e.g., URL *u)* **into one of the caches**
- Use a hash function that **maps *u*** to node ***h(u)***
  - For example, ***h(x) = (ax + b) mod p*,** where *p* is range of *h(x), i.e.,* the number of caches
  - Interpret *u* as a number based on bit pattern of object ID (or URL)
- Hashing tends to **distribute input uniformly** across range of hash function
  - Objects (URLs) are **equally balanced** across caches, even if object IDs are skewed (i.e., highly clustered in ID space)
- No one cache responsible for an **uneven share of objects/URLs**
- No disproportionately loaded node (potential bottleneck)

$h(u) = (7u + 4) \bmod 5$

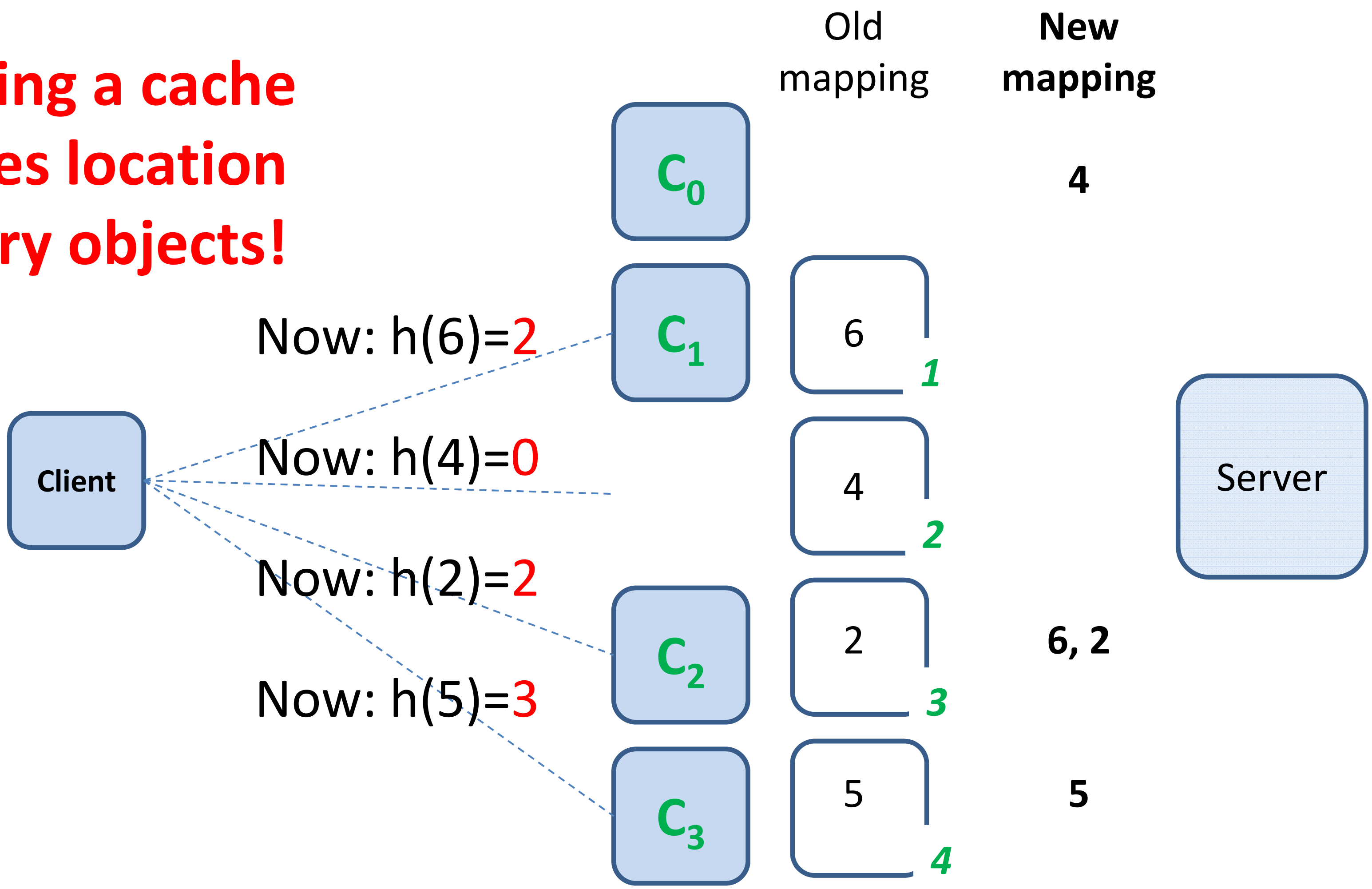
Assume, we have **five caches**, numbered 0, …, 4.



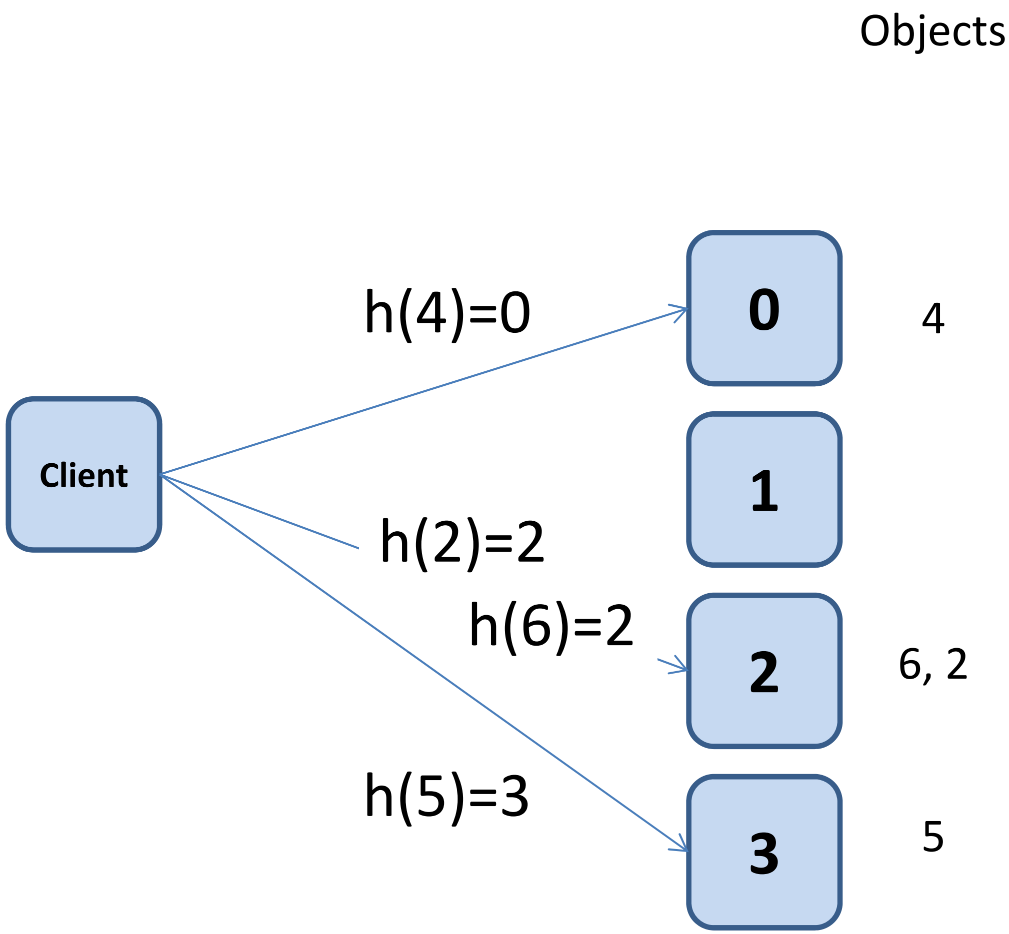
$h(u) = (7u + 4) \bmod 4$

(now have to map across 4 caches)

Removing a cache changes location of every objects!

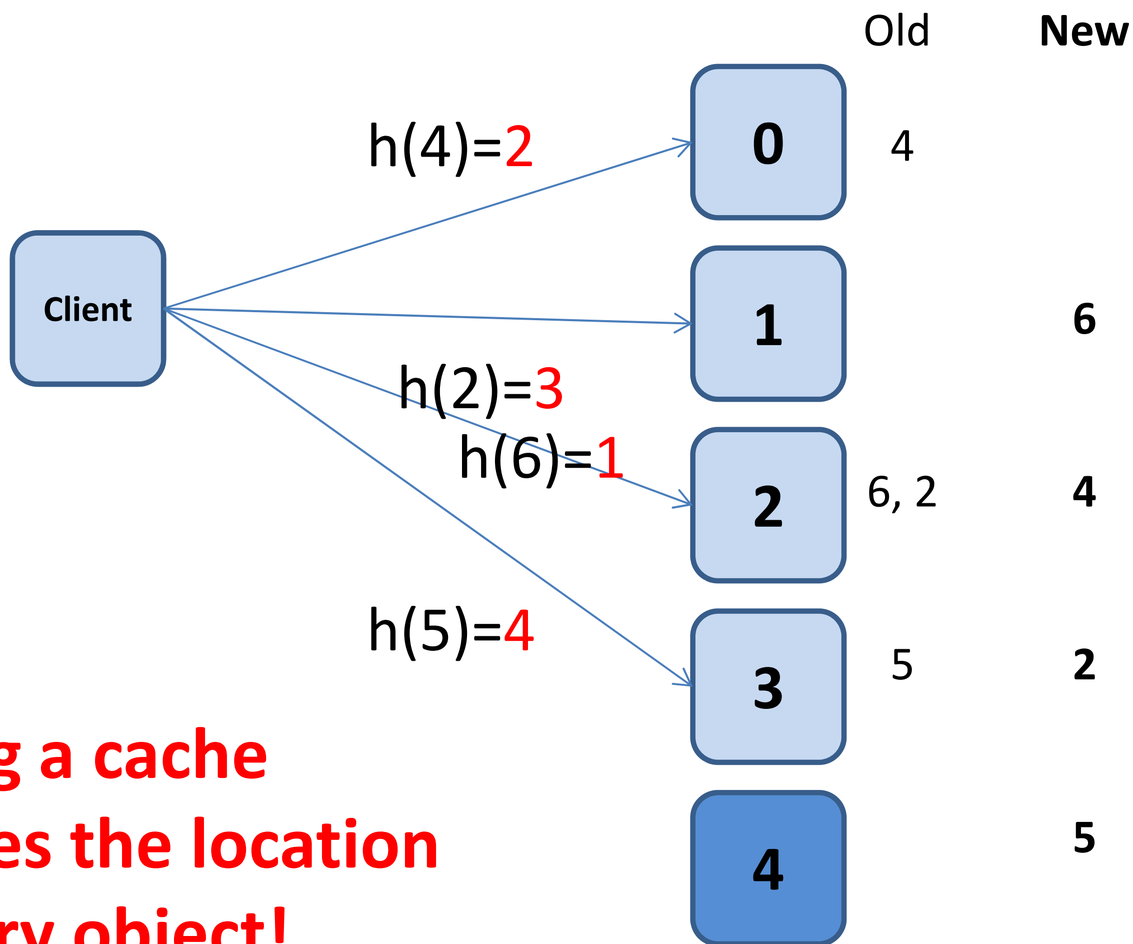


$h(u) = 7u + 4 \bmod 4$
(mapped across 4 nodes)



$h(u) = (7u + 4) \bmod 5$

(adding a cache again)



Adding a cache changes the location of every object!

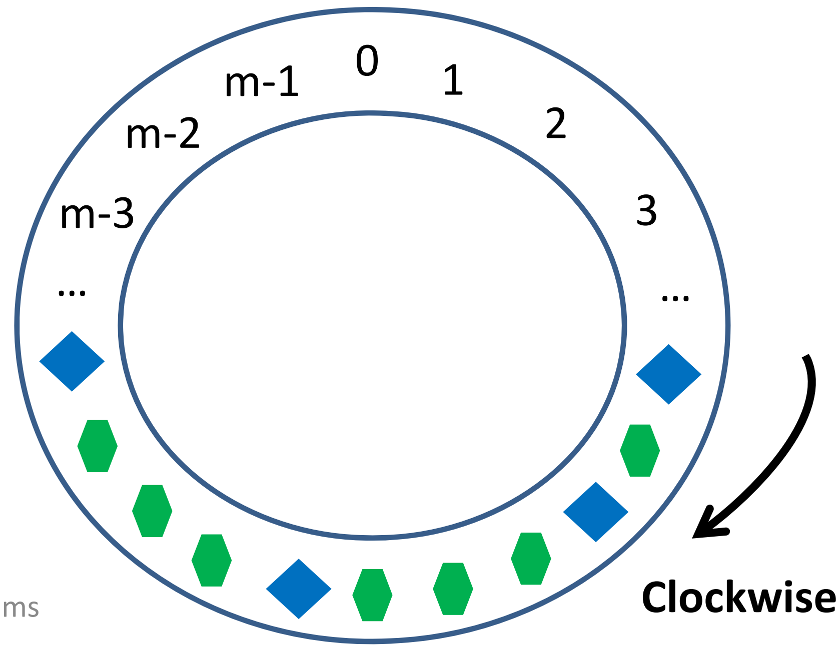
Consistent hashing

- **Goals**
  - Uniform distribution of objects across nodes
  - Easily find objects
  - Let any client perform a local computation mapping a URL to node that contains referenced object
  - **Allow for nodes to be added/removed without much disruption**
- D. Karger *et al.*, MIT, 1997
- Basis for Akamai
  - CDN company (content distribution network)
  - Web cache as a service

Consistent hashing

Key idea intuition

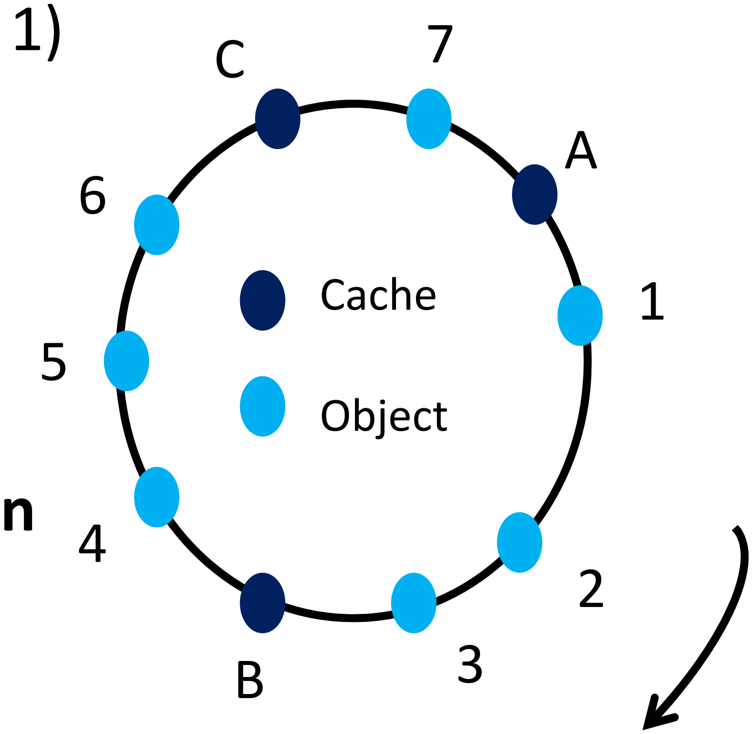
- Select a **base hash function** that maps input identifier to the number range [0, …, m-1]
- *E.g.*, $\boldsymbol{h(x) = (ax + b) \bmod m}$
- Interpret range of h(..) as array that wraps around (i.e., a circle)
- h(..) gives slot in array (circle) and wraps around at m-1 to 0
- Each **object is mapped to a slot** via h(..)
- Each **cache is mapped to a slot** via h(..)
- Assign **each object to the closest cache slot** in **clockwise direction** on the circle



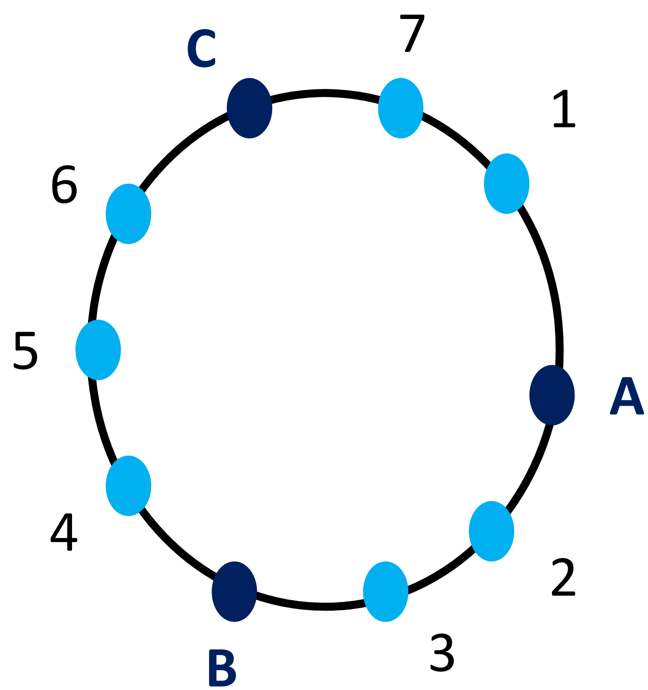
Consistent hashing

Original interpretation

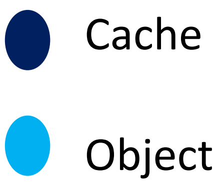
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle via *h*(..)
- **Each cache is mapped to a point** on unit circle via *h*(..)
- Assign **each URL to closest cache point** in **clockwise direction** on the circle



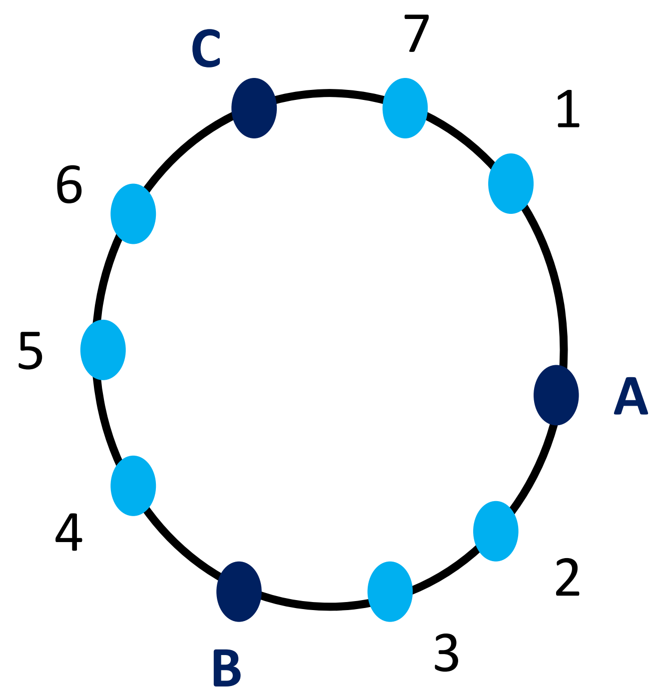
Mapping items to caches



Items	2, 3	mapped to **B**
Items	4, 5, 6	mapped to **C**
Items	7, 1	mapped to **A**

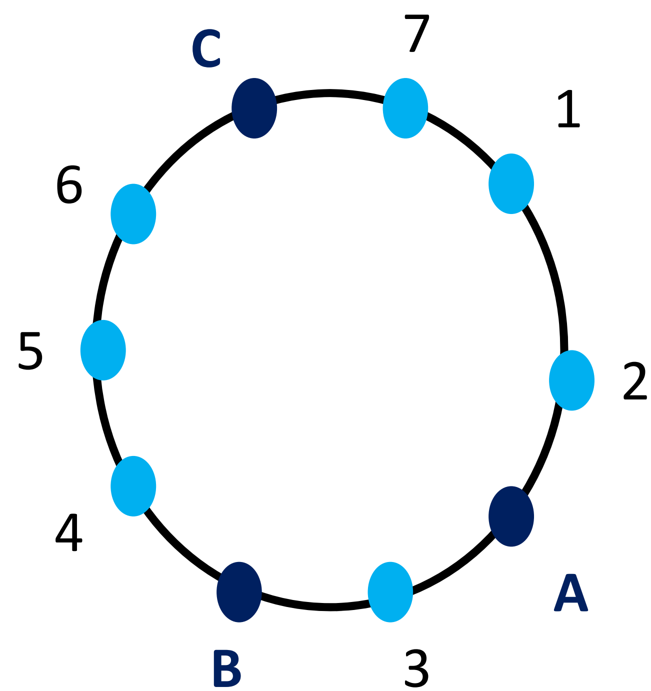


Removing a cache



Items	2, 3, **7, 1**	mapped to	**B**
Items	4, 5, 6	mapped to	**C**
Items	7, 1	mapped to	**A**

Adding a cache

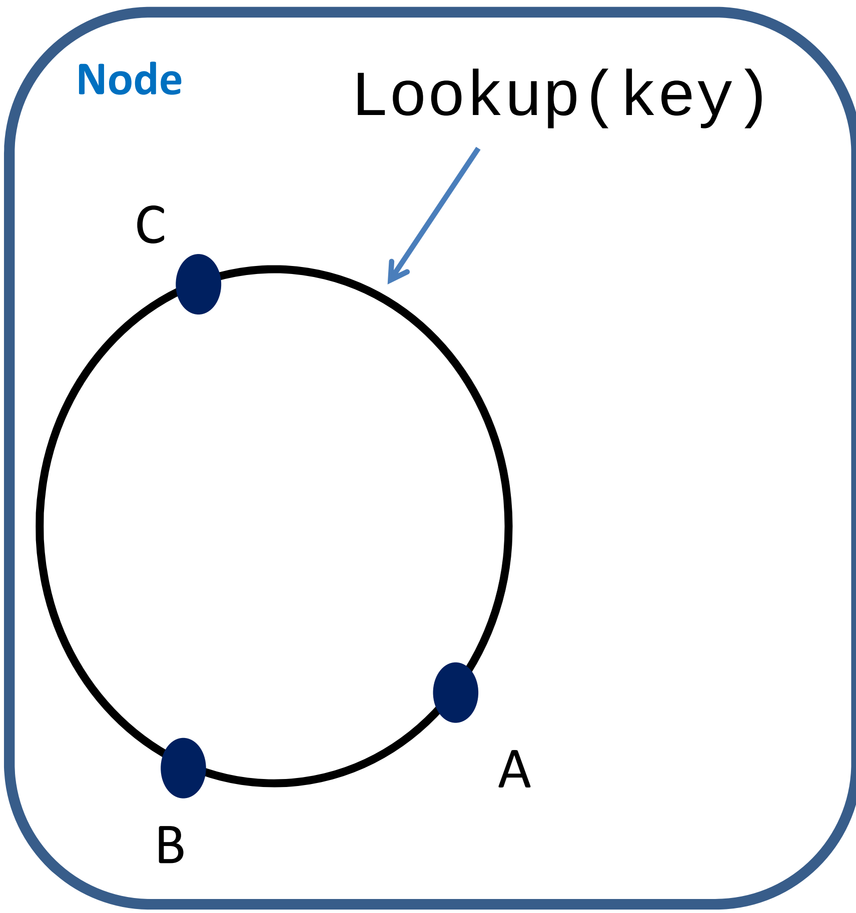


Items 7, 1, 2, 3 **mapped to B**

Items 4, 5, 6 mapped to **C**

Items **7, 1, 2** mapped to **A**

Processing a Lookup(key)



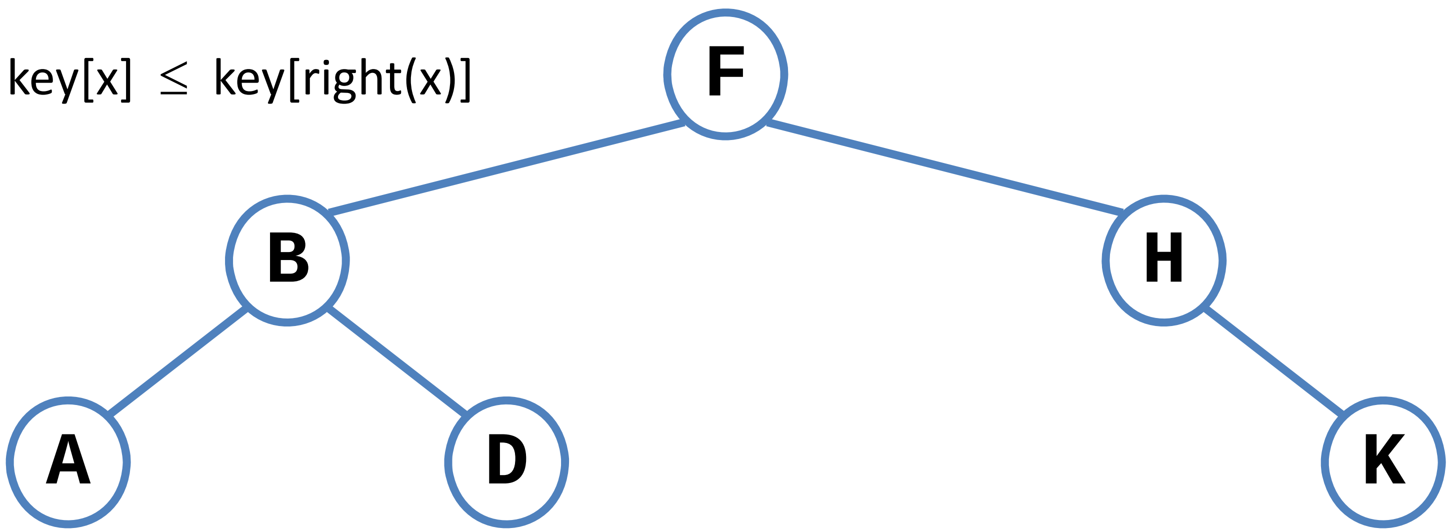
Retrieve ***object*** with ***key*** from ***A***

Information about **node addition** & **removal** (e.g., via gossiping or via a coordination service)

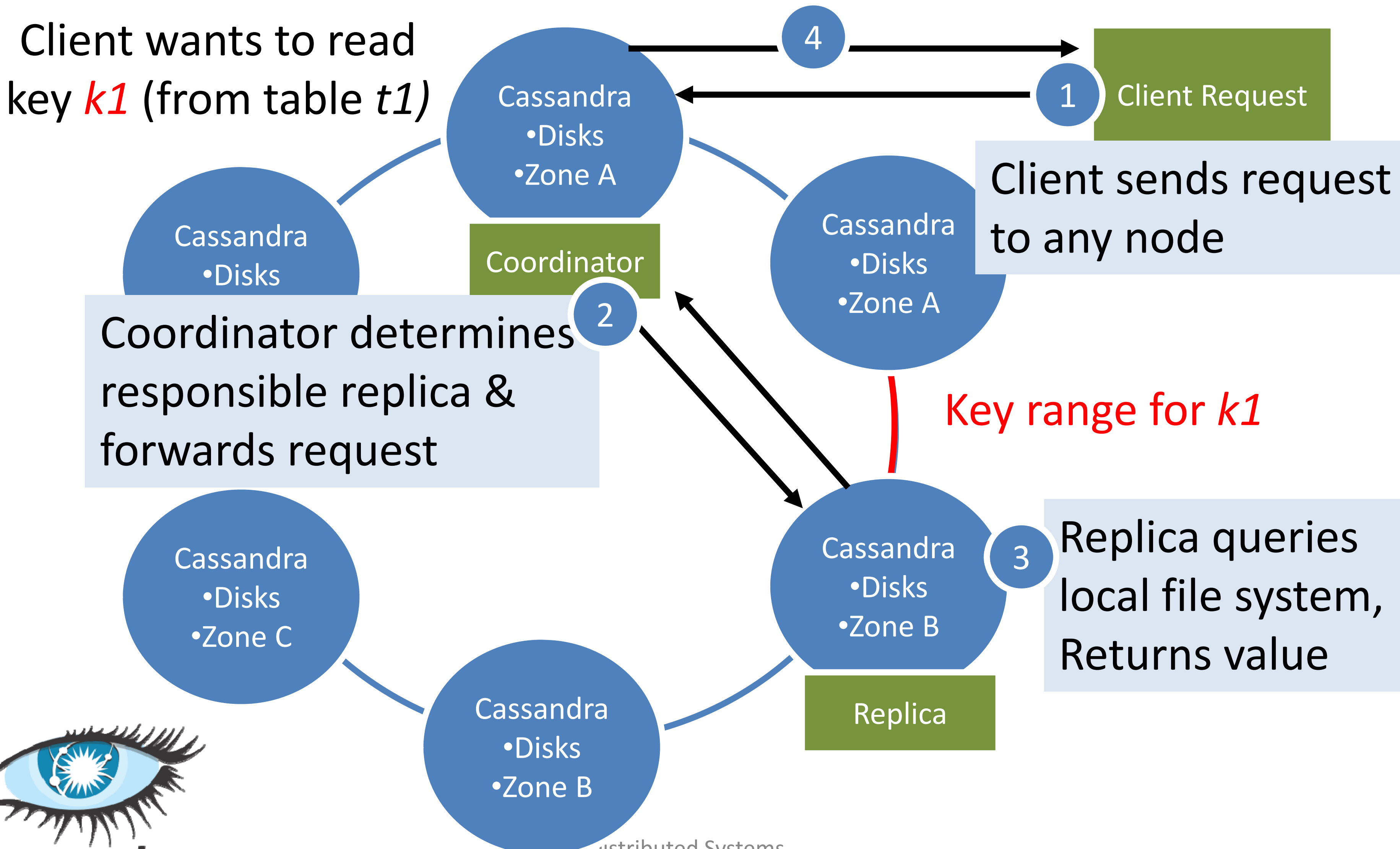
Cache lookup data structure at each node

- Store **cache points** in a **binary tree**
- Find **clockwise successor of a URL point** by single search in tree (takes **O(log n) time**)
- For a constant time technique, cf. Karger et al., 1997

key[left(x)] $\leq$ key[x] $\leq$ key[right(x)]



Cassandra global read-path



Base hash function: MD5

- **Message Digest 5** (MD5), R. Rivest, **1992** (MD1, ..., MD6)
- **Hash function** that produces a **128-bit** (16-byte) **hash value**
- Maps variable-length message into a **fixed-length output**
- MD5 hash is typically expressed as a hex number (32 digits)
- It's been shown that **MD5 is not collision resistant**
- US-CERT about MD5 "***should be considered cryptographically broken and unsuitable for further use***" (for security, not for caching)
- SHA-2 is a more appropriate cryptographic hash function
- For consistent hashing, MD5 is sufficient

MD5 examples

- MD5(“*The quick brown fox jumps over the lazy dog*”) = 9e107d9d372bb6826bd81d3542a419d6
- MD5(“*The quick brown fox jumps over the lazy dog***.**”) = e4d909c290d0fb1ca068ffaddf22cbd0
- MD5(“”) = d41d8cd98f00b204e9800998ecf8427e

http://en.wikipedia.org/wiki/MD5#Algorithm