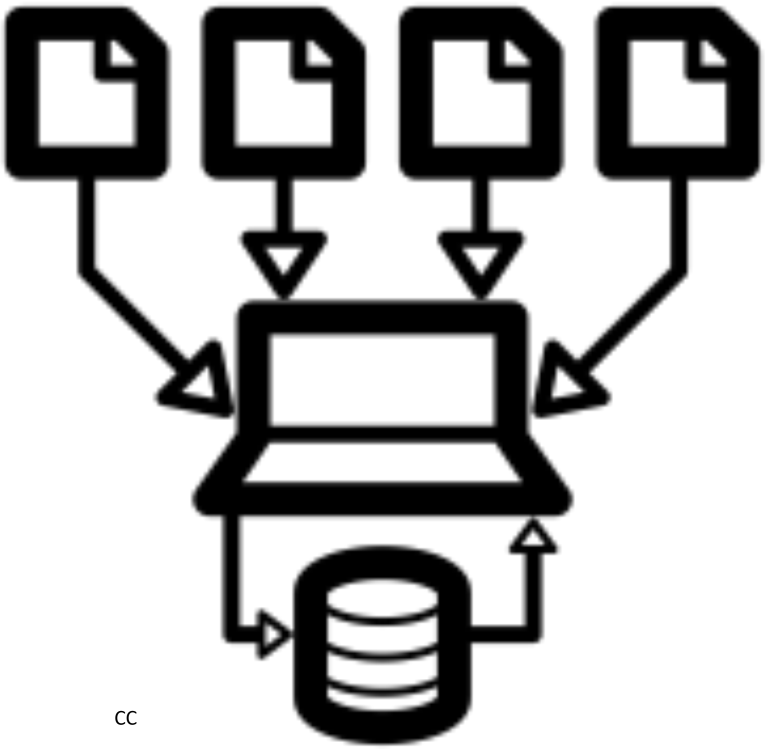



# Massively Available Key-Value Stores & Consistent Hashing

# Key-value stores

- *What is a key-value-store?*
- *Why are key-value stores needed*?
- Key-value-store client interface
- Key-value stores in practice
- Common features & non-features

***What mechanisms make them work?***

# *What are key-value stores?*

- Container for key-value pairs (databases)
- Distributed, multi-component, systems
- NoSQL semantics (non-relational)
- KV-stores offer **simpler query semantics** in exchange for **increased scalability**, **speed**, **availability**, and **flexibility**
- Data model not new

# DBMS (SQL)

# Key-value store

**Students Table**

| Student | ID* |
|---|---|
| John Smith | 084 |
| Jane Bloggs | 100 |
| John Smith | 182 |
| Mark Antony | 219 |

**Activities Table**

| ID* | Activity* | Cost |
|---|---|---|
| 084 | Swimming | $17 |
| 084 | Tennis | $36 |
| 100 | Squash | $40 |
| 100 | Swimming | $17 |
| 182 | Tennis | $36 |
| 219 | Golf | $47 |
| 219 | Swimming | $15 |
| 219 | Squash | $40 |

- Relational data schema
- Data types
- Foreign keys
- Full SQL support

| Key | Value |
|---|---|
| John Smith | {Activity:Name= Swimming} |
| Jane Bloggs | {Activity:Cost=57} |
| Mark Anthony | {ID=219} |

- No data schema
- Raw byte access
- No relations
- Single-row operations

# *Why are key-value stores needed?*

- Today's internet applications
  - Huge amounts of stored data
  - Huge number of Internet users
  - Frequent updates
  - Fast retrieval of information
  - Rapidly changing data definitions
- Ever more users, ever more data

# *Why are key-value stores needed?*

- Horizontal scalability
  - User growth, traffic patterns change
  - Adapt to number of requests, data size
- Performance
  - High speed for single-record read and write operations
- Flexibility
  - Adapt to changing data definitions

# *Why are key-value stores needed?*

- Reliability
  - Thousands of components at play
  - Uses commodity hardware: failure is the norm
  - Provide failure recovery
- Availability and geo-distribution
  - Users are worldwide
  - Guarantee fast access

# Key-value store client interface

- Main operations
  - Write/update `put(key, value)`
  - Read `get(key)`
  - Delete `delete(key)`

- Usually no aggregation, no table joins, no transactions!

# Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

# Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

Initialization Using ZooKeeper

# Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1
put.add(Bytes.toBytes("colfam1"), Byt ... toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value"));
System.out.println("Value: " + Bytes.toInt(val));
```

Initialization
Using
ZooKeeper

Column Family:
"Schema"

# Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");

HTable table = new HTable(conf, „MyBaseTable");

Put put = new Put(Bytes.toBytes("key1"
put.add(Bytes.toBytes("colfam1"), Byt    toBytes(„value"), Bytes.toBytes(200));
table.put(put);

Get get = new Get(Bytes.toBytes(“key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), B
System.out.println("Value: " + Bytes.toInt(val));
```

Initialization Using ZooKeeper

Column Family: “Schema”

Column: Defined at run-time ( “wide column” stores)

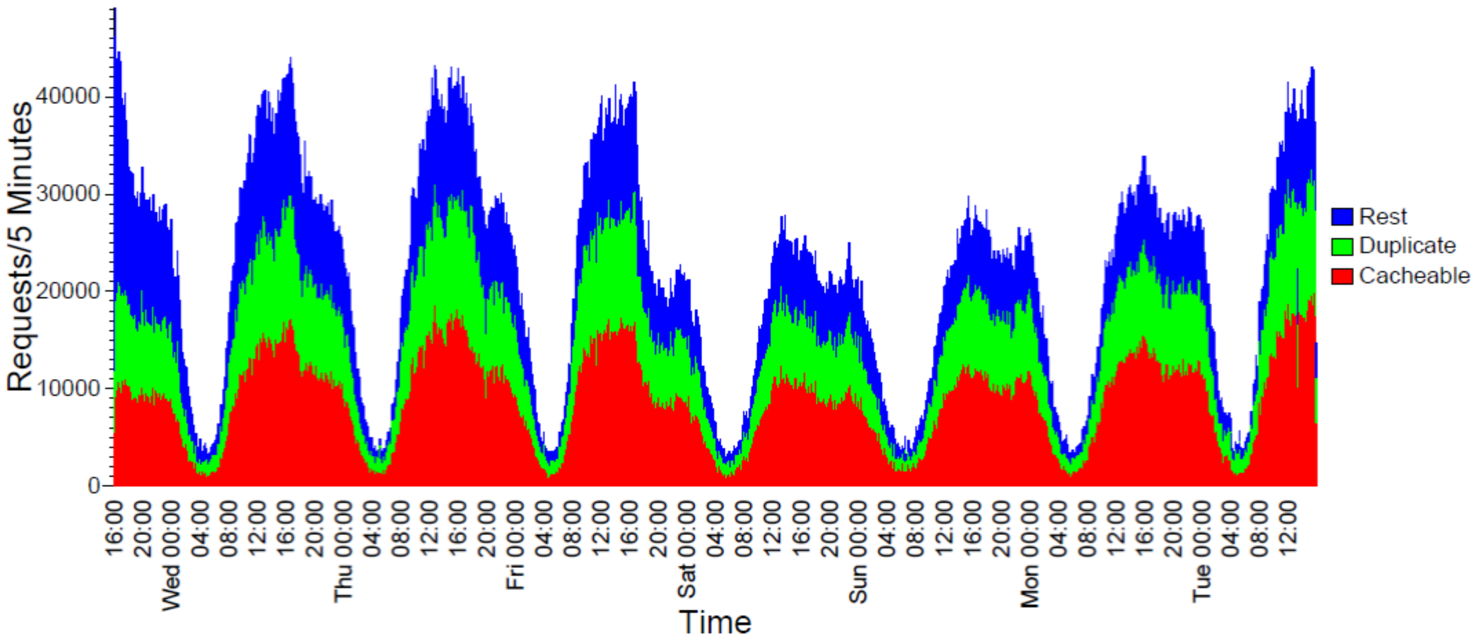
# Key-value store in practice

- BigTable
- Apache HBase
- Apache Cassandra
- Redis
- Amazon Dynamo
- Yahoo! PNUTS

# Common elements of key-value stores

- Failure detection, failure recovery
- Replication (*cf. Replication*)
  - Store and manage multiple copies of data
- Versioning (*cf. Time*)
  - Store different versions of data
  - Timestamping

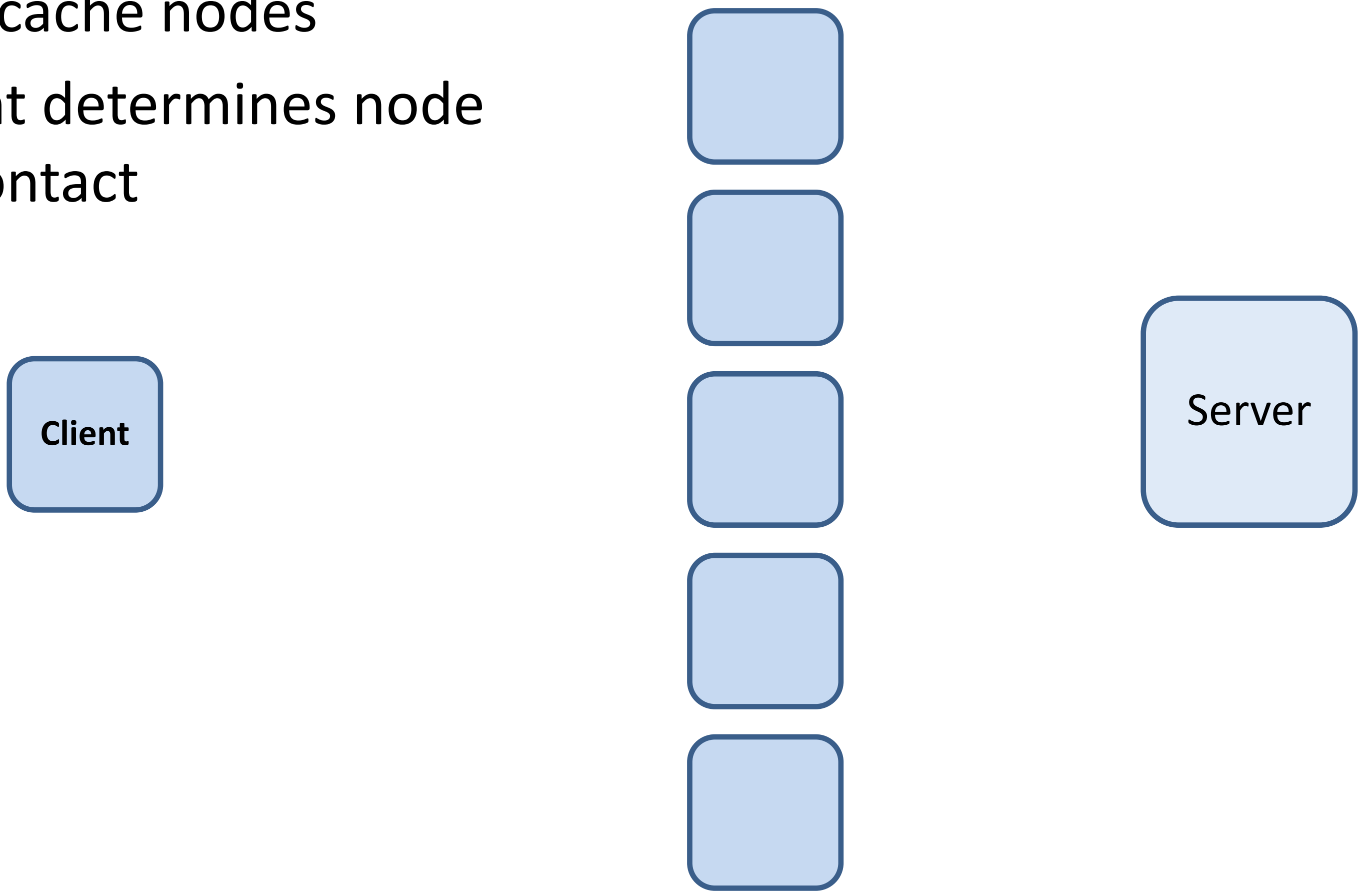
# Consistent Hashing




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

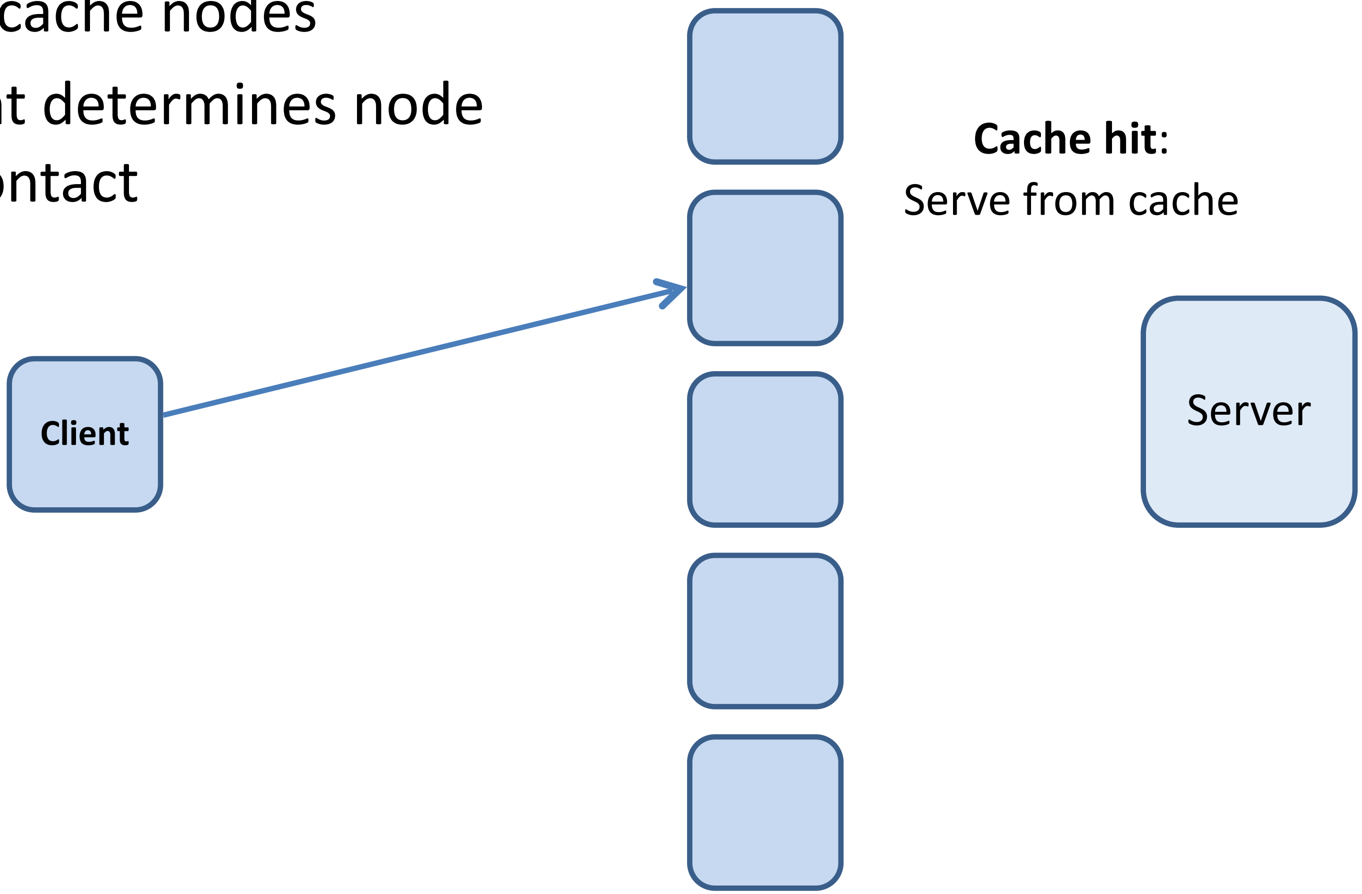
# Caching

- Five cache nodes
- Client determines node to contact



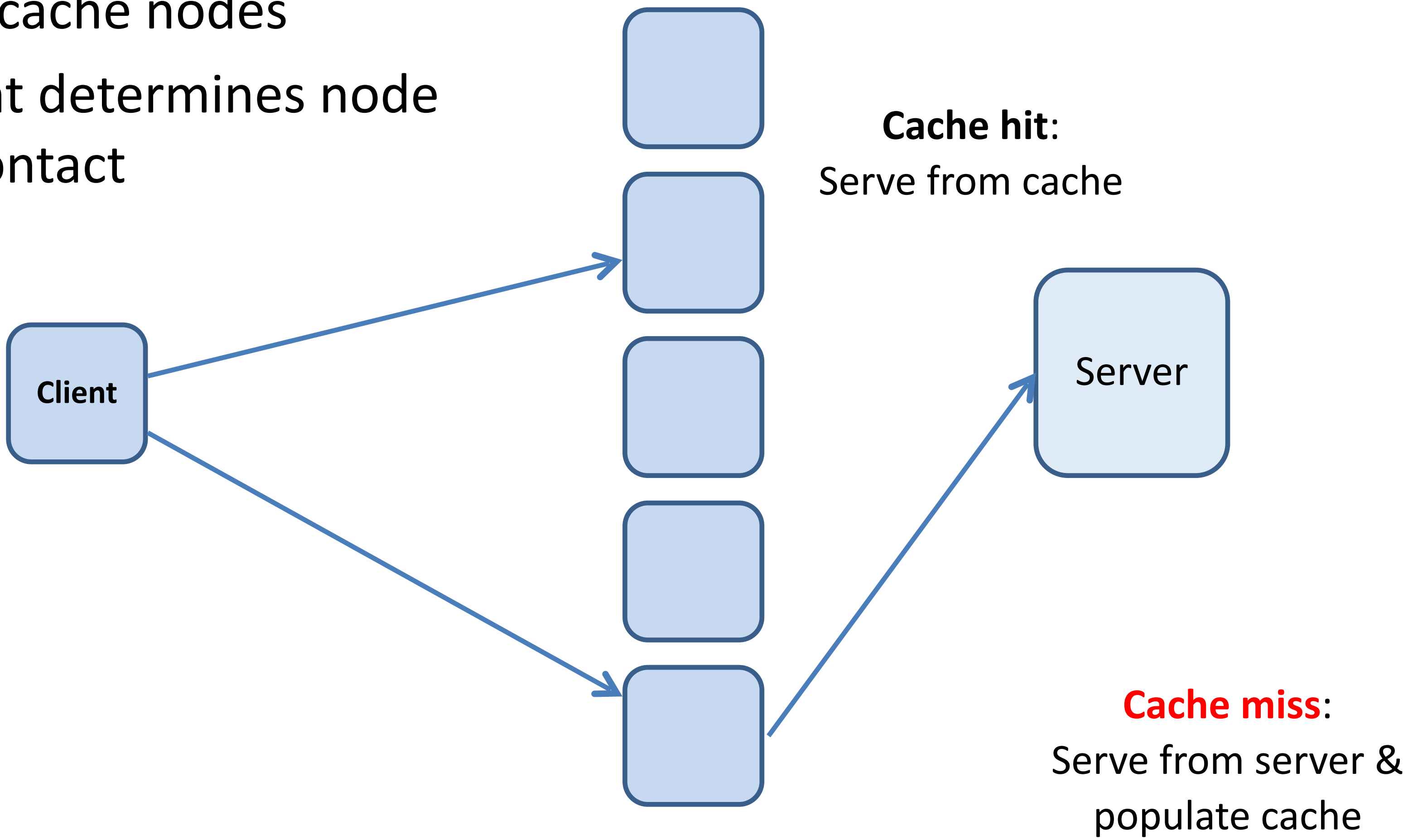
# Caching

- Five cache nodes
- Client determines node to contact



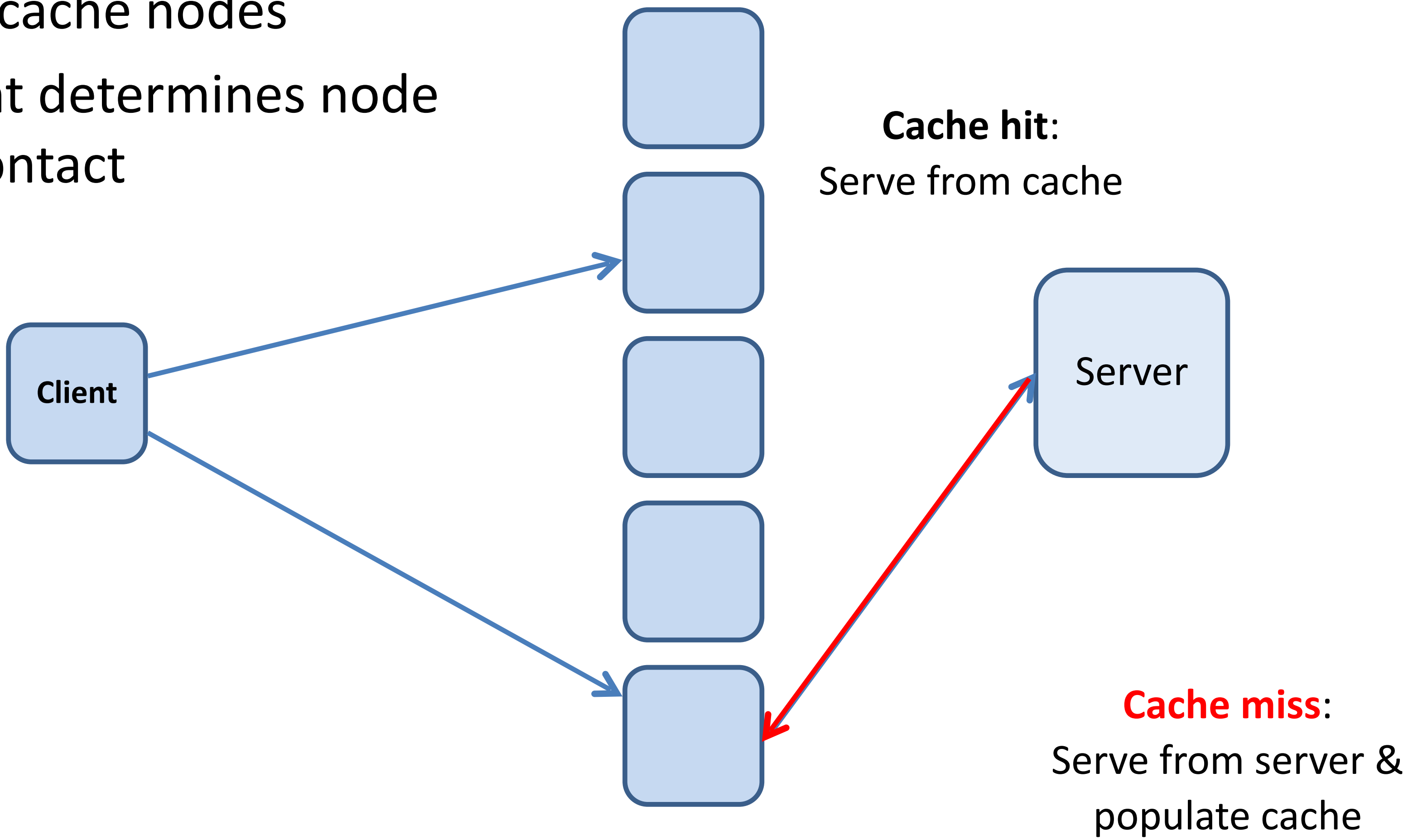
# Caching

- Five cache nodes
- Client determines node to contact



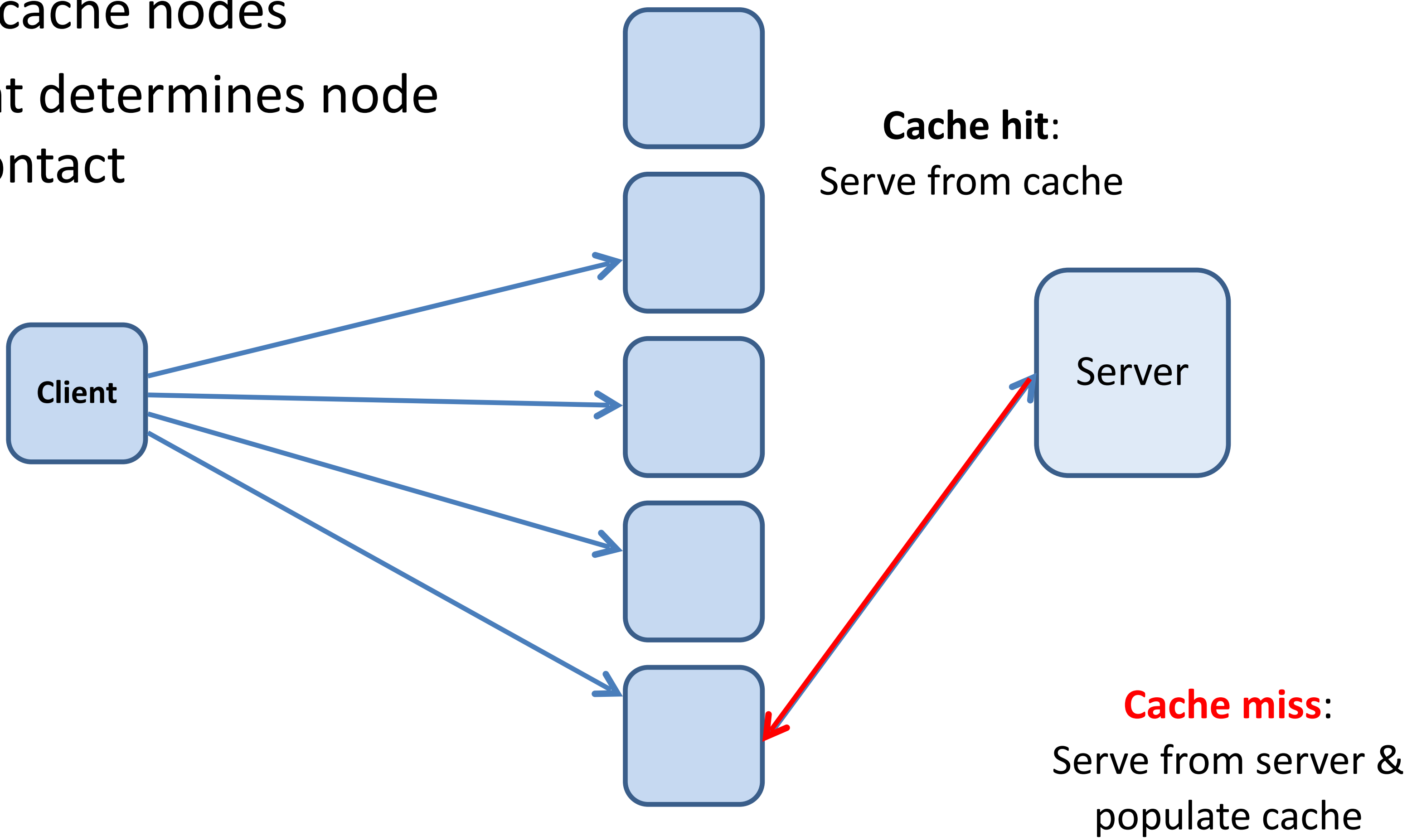
# Caching

- Five cache nodes
- Client determines node to contact



# Caching

- Five cache nodes
- Client determines node to contact



# Problem: Mapping objects to caches

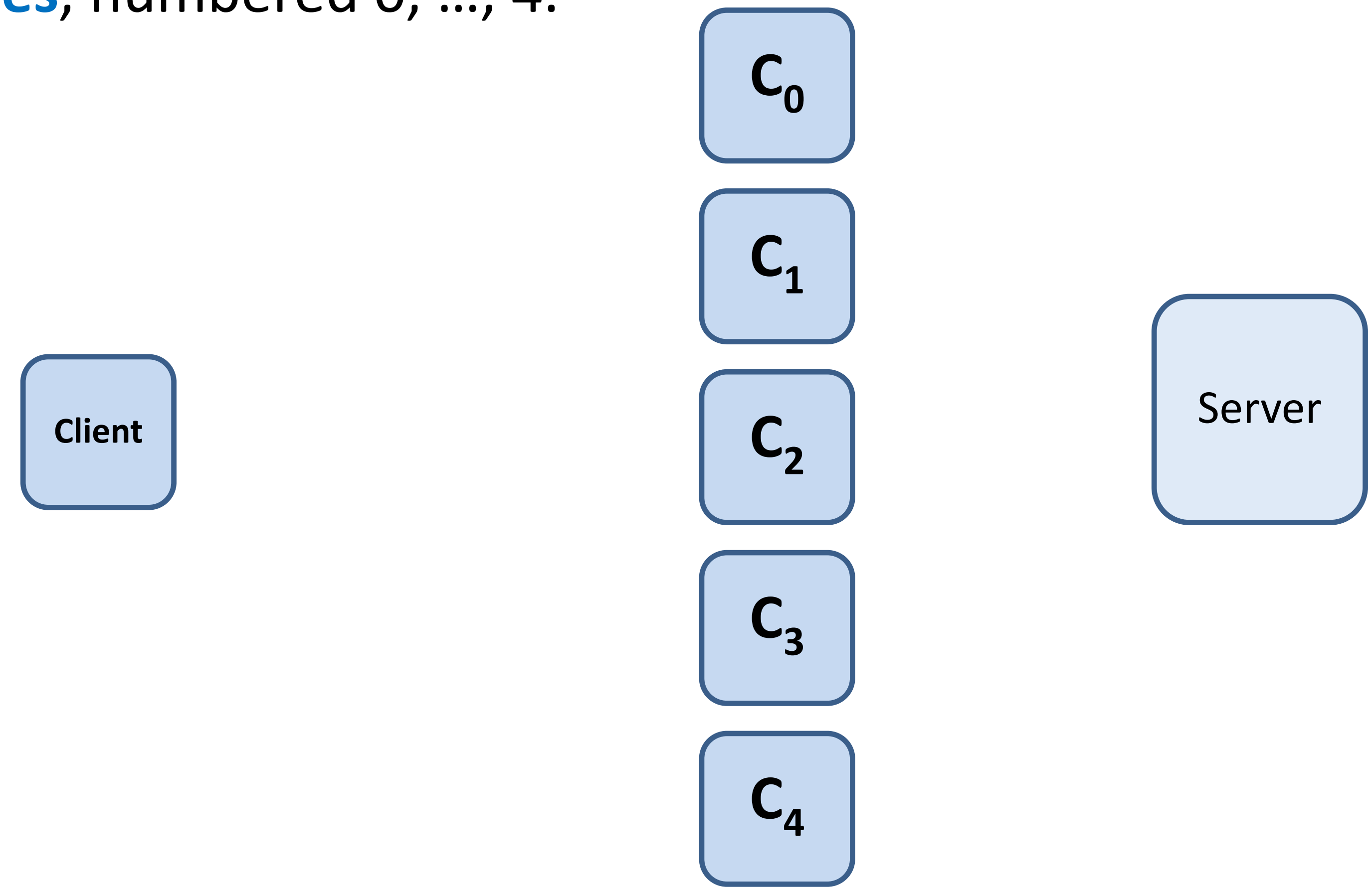
- Given a number of caches (e.g., cooperative caching, CDNs, etc.)
- Each cache should carry an **equal share of objects**
- **Clients** need to **know what cache to query** for a given object
- **Horizontally partition** (shard) object ID space
  - **Doesn’t work** with **skewed distributions**: e.g., 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)

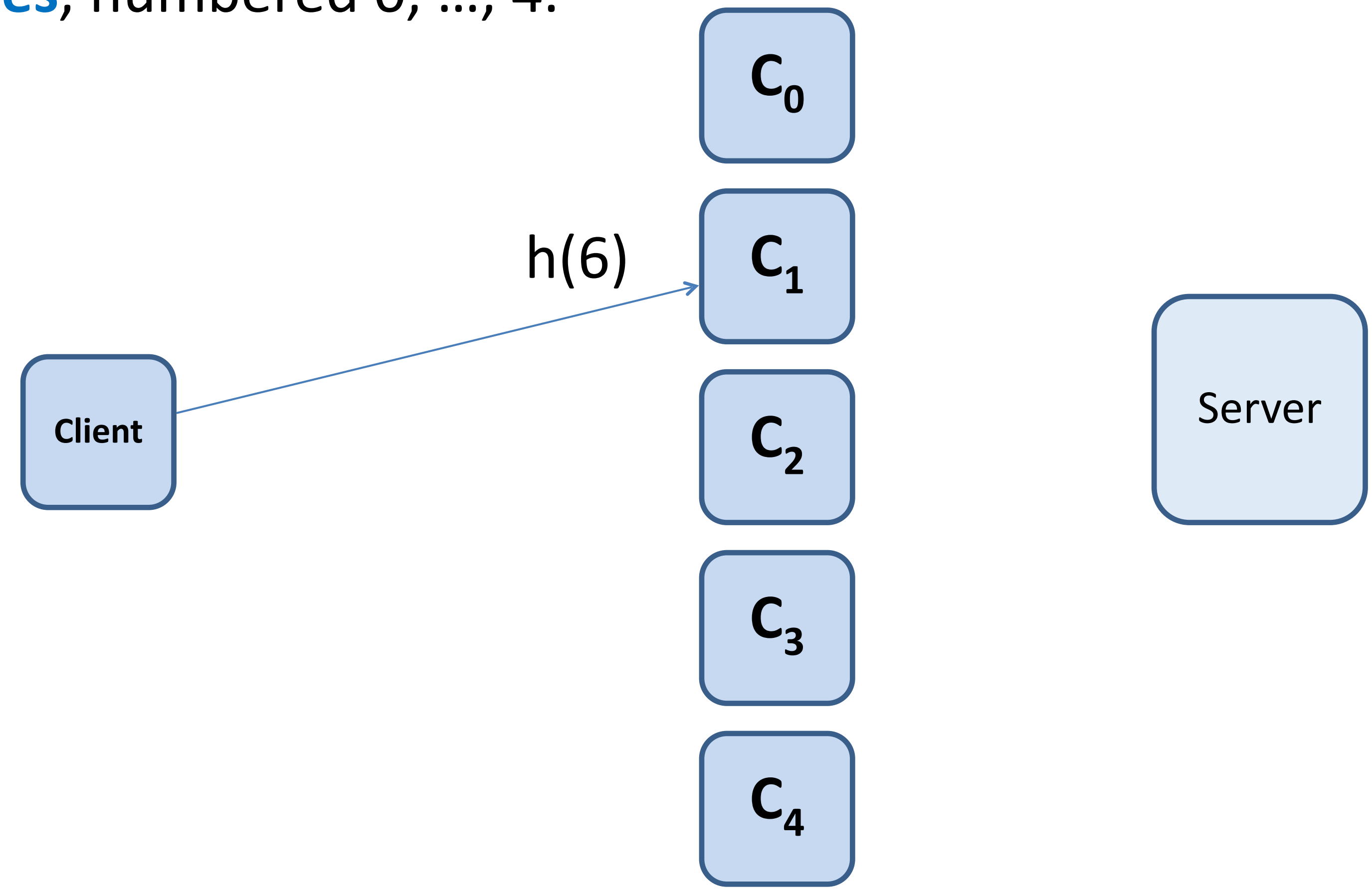
# h(u) = (7u + 4) mod 5

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



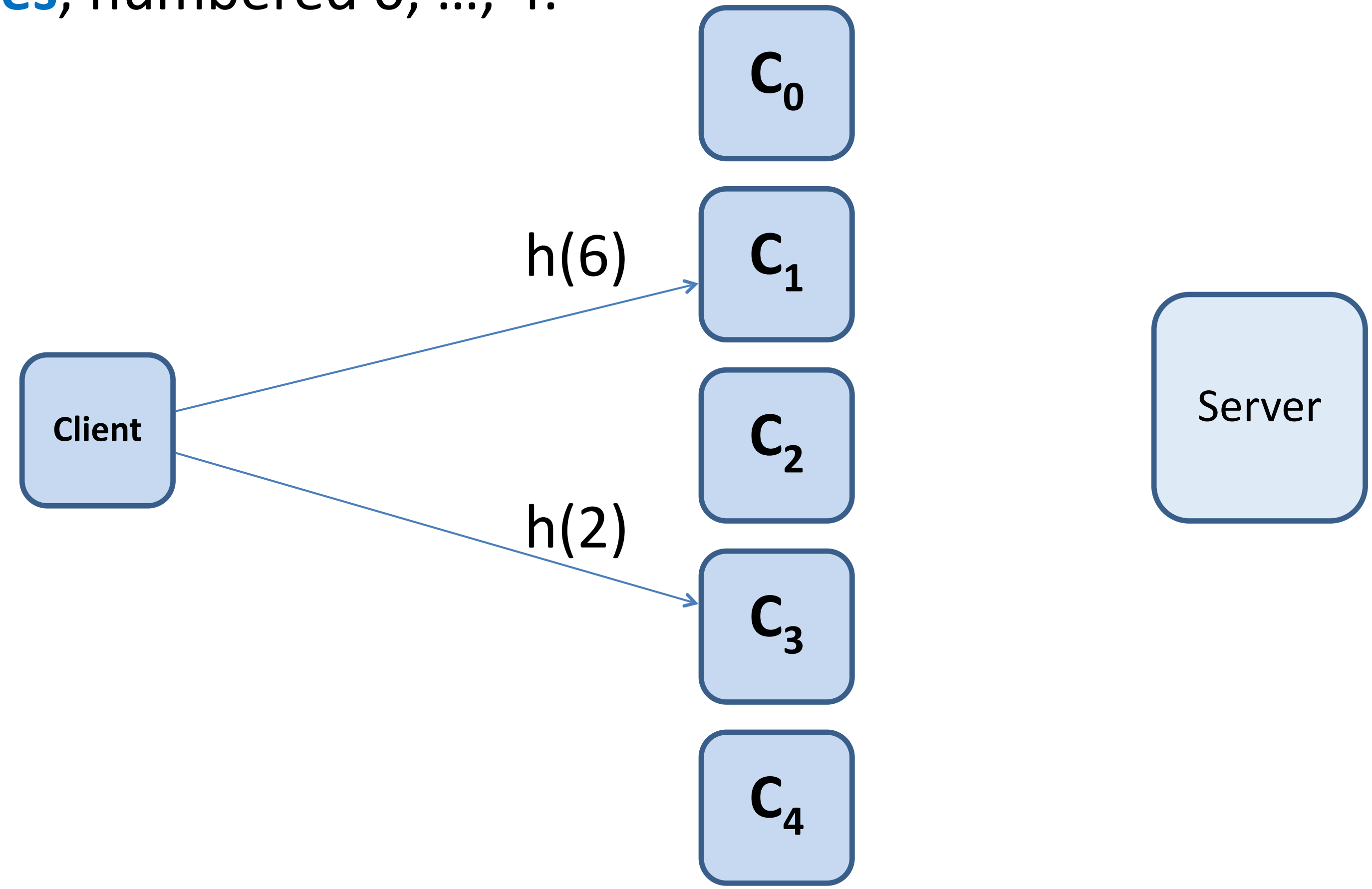
## $h(u) = (7u + 4) \bmod 5$

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



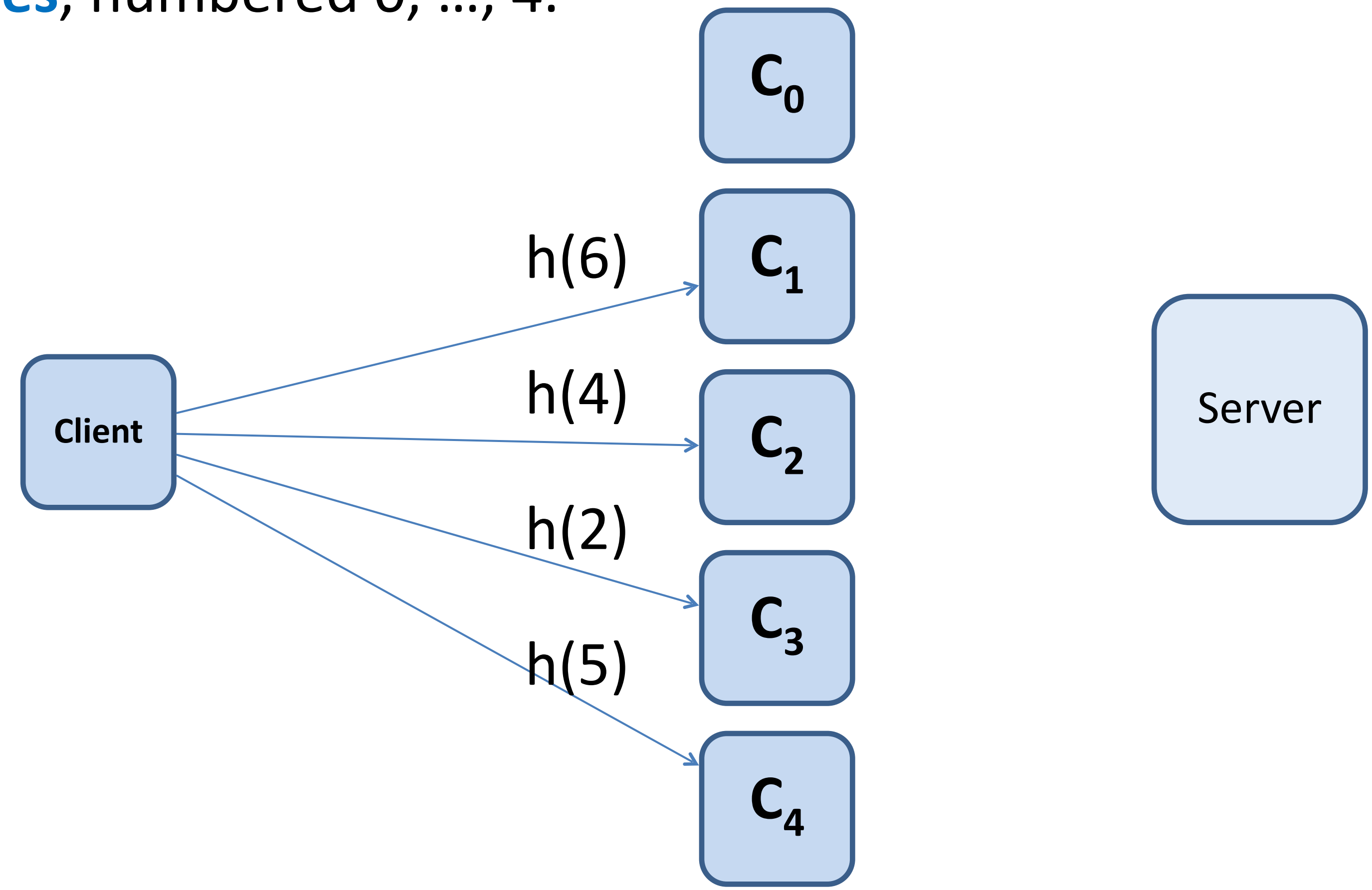
# $h(u) = (7u + 4) \bmod 5$

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



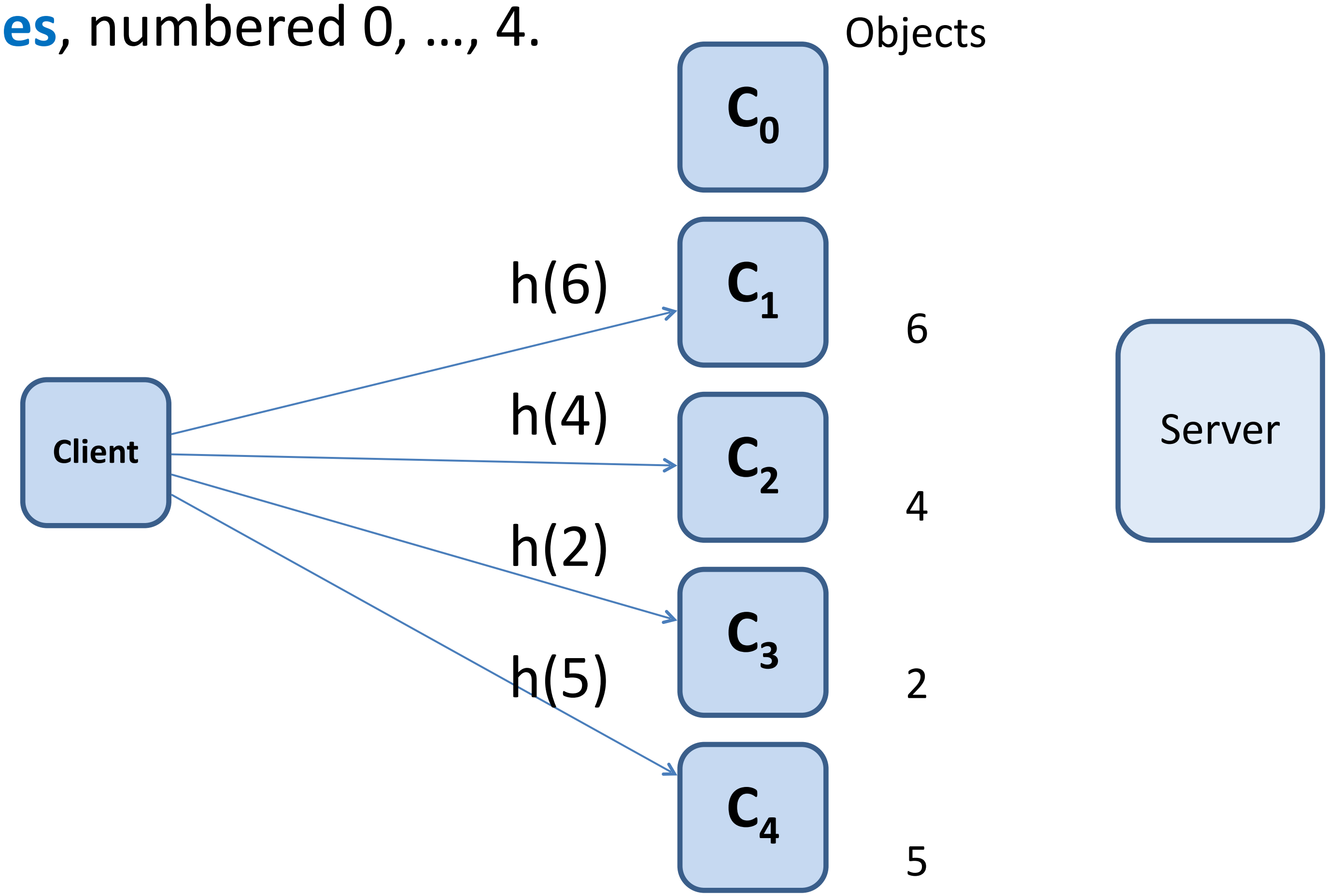
# $h(u) = (7u + 4) \bmod 5$

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



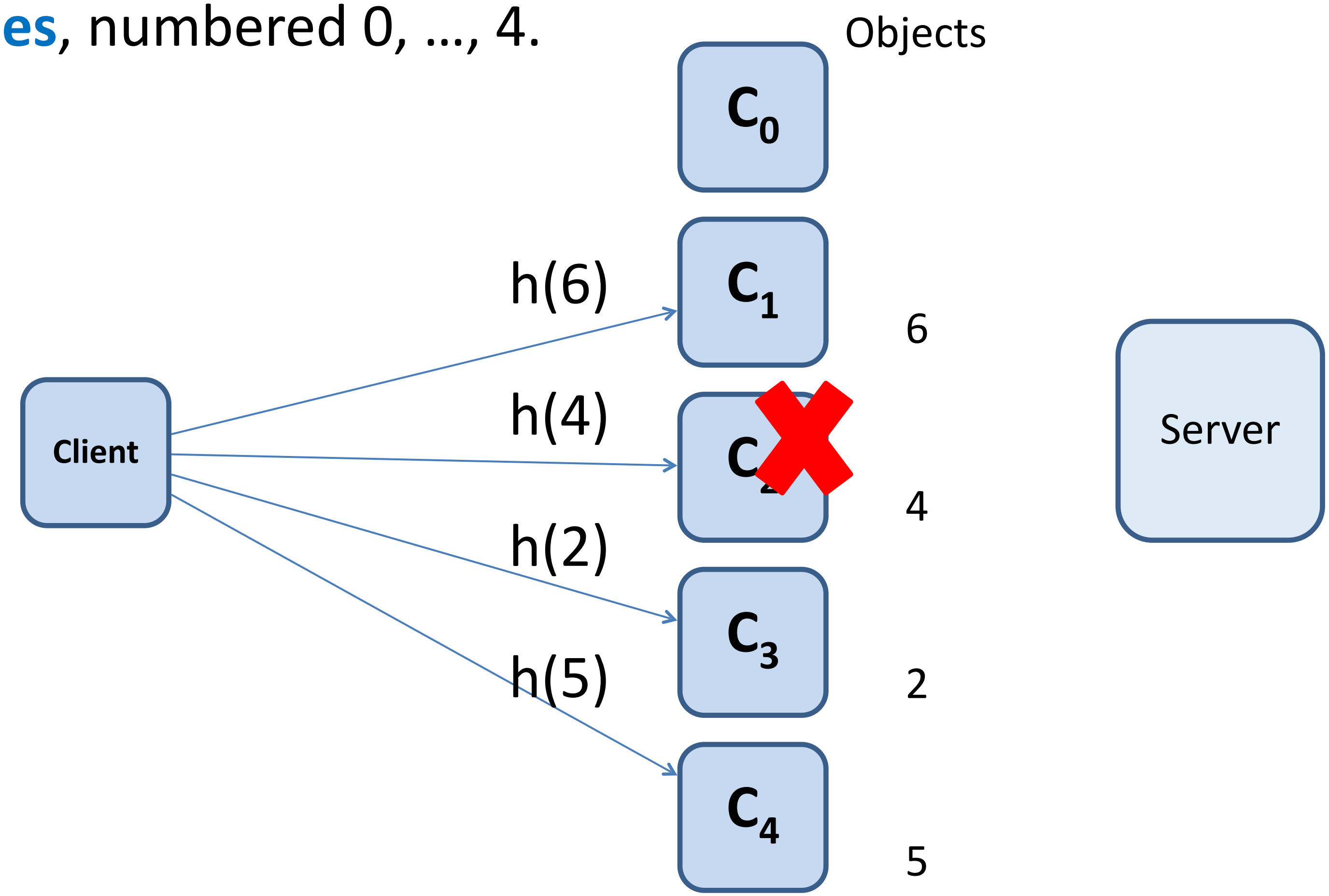
# $h(u) = (7u + 4) \bmod 5$

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



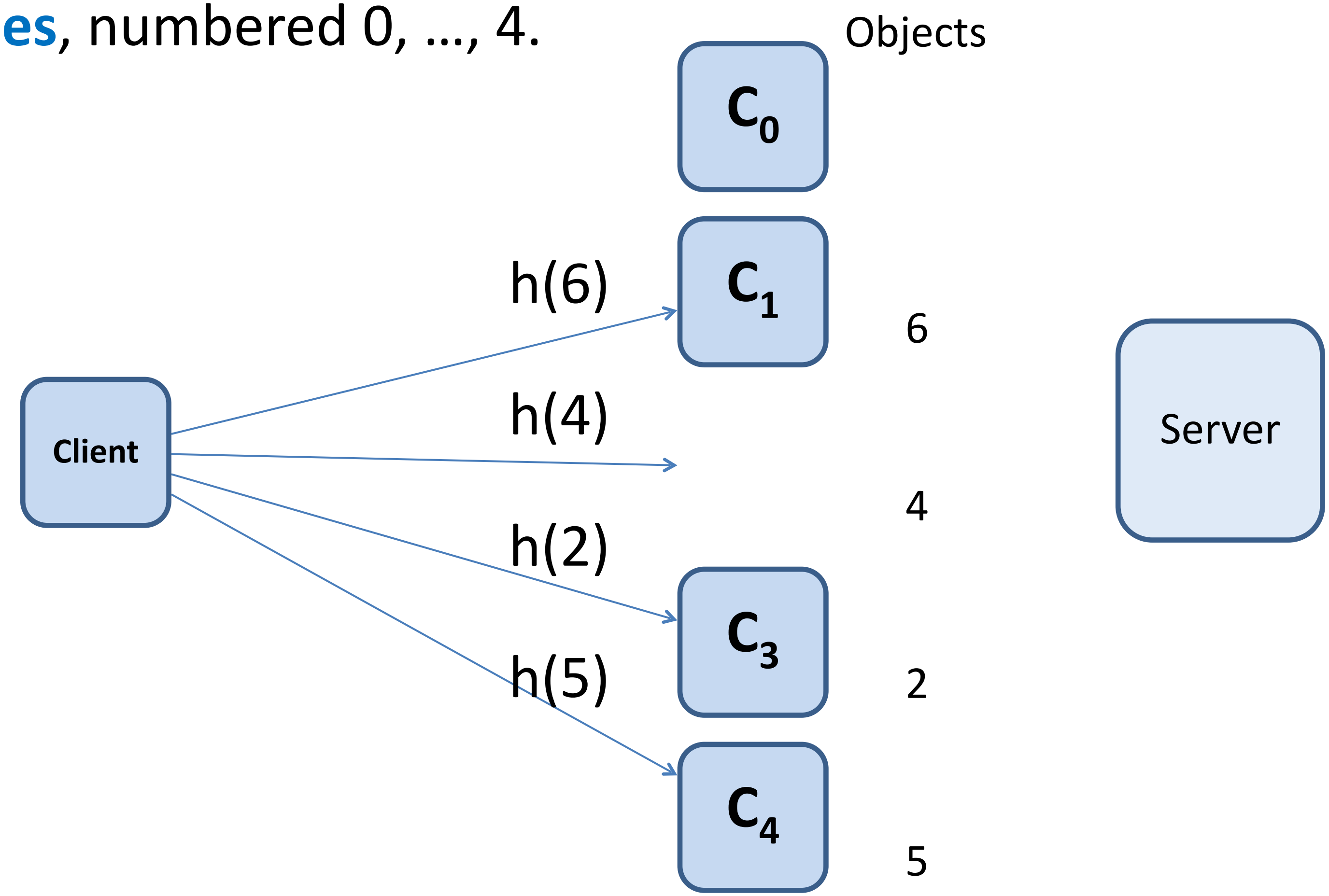
# $h(u) = (7u + 4) \bmod 5$

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



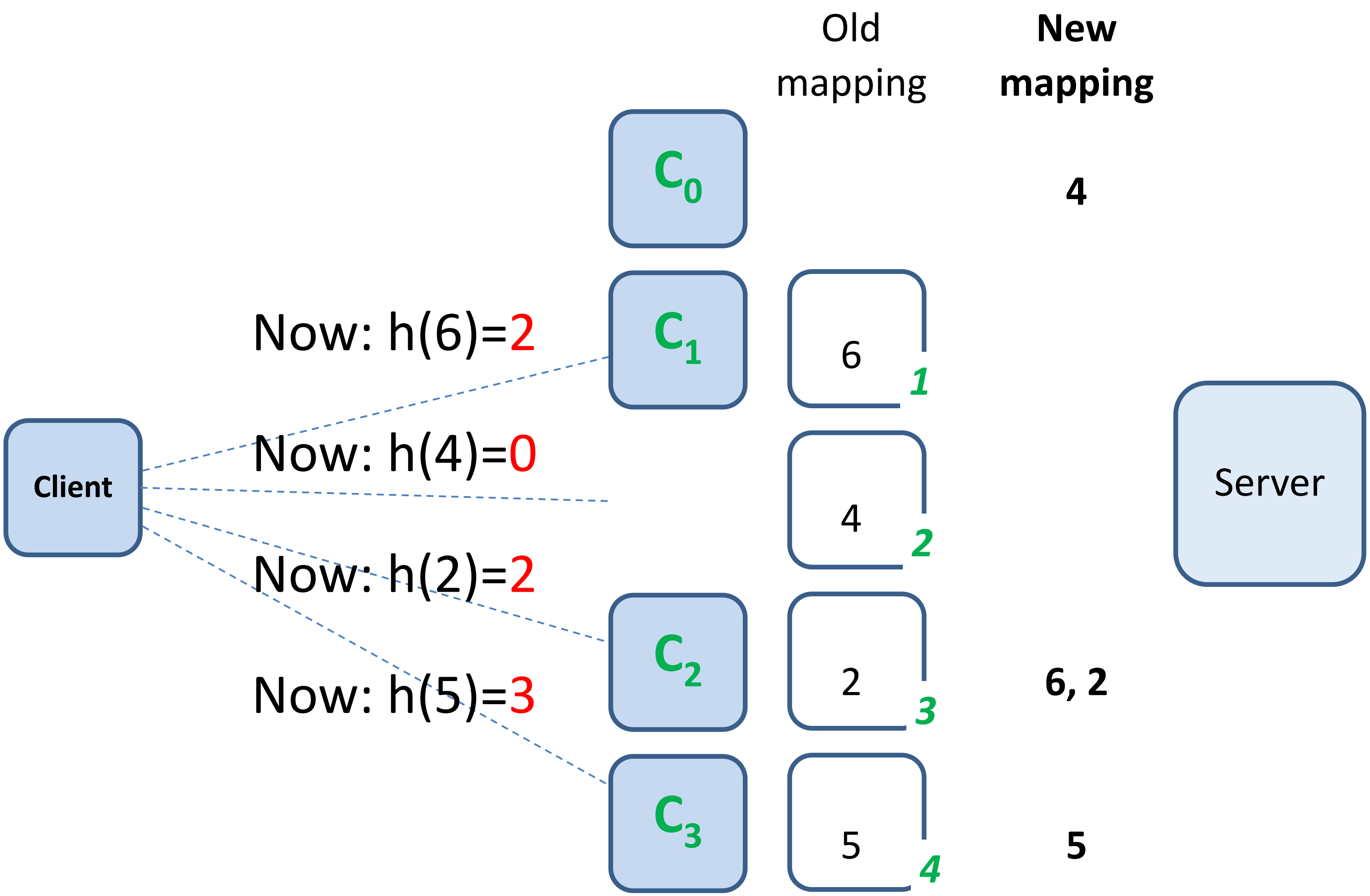
# h(u) = (7u + 4) mod 5

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



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

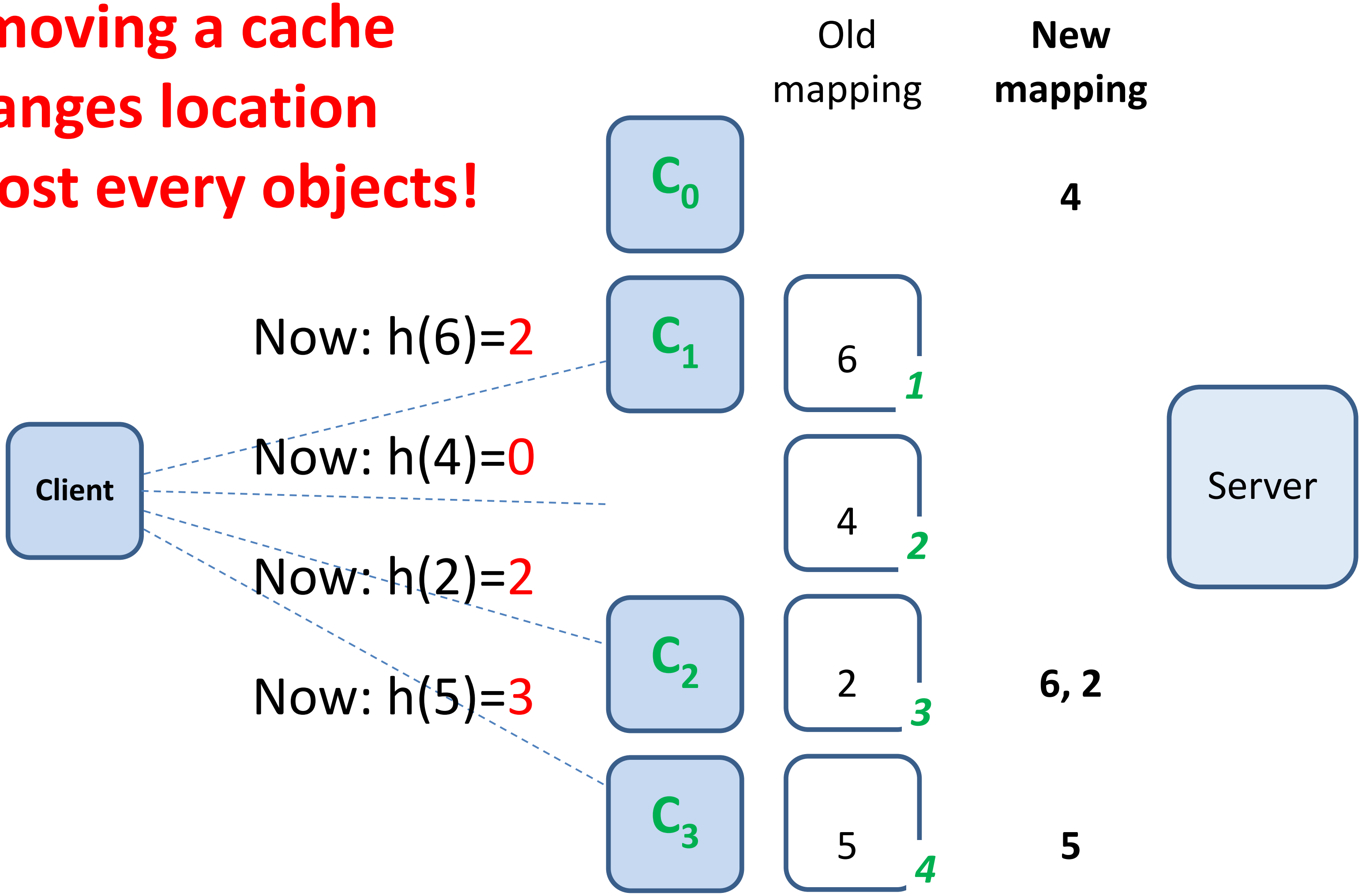
## (now have to map across 4 caches)



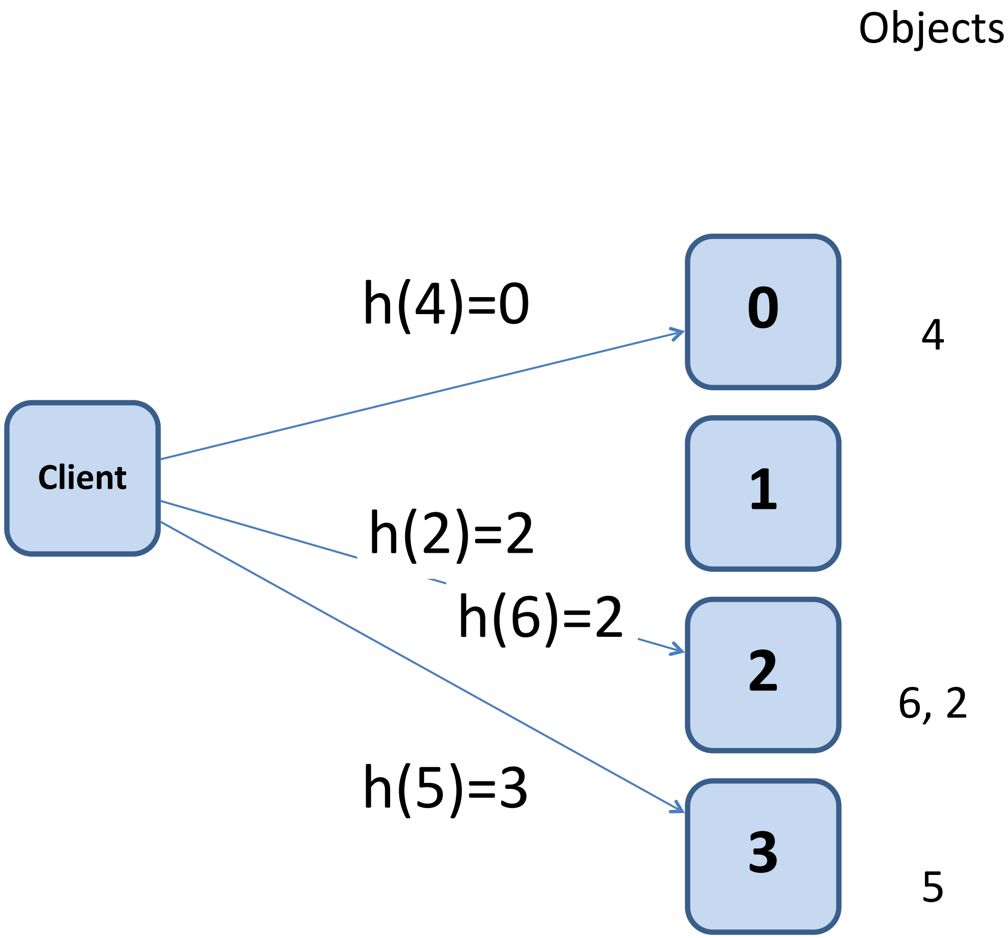
# $h(u) = (7u + 4) \bmod 4$

**(now have to map across 4 caches)**

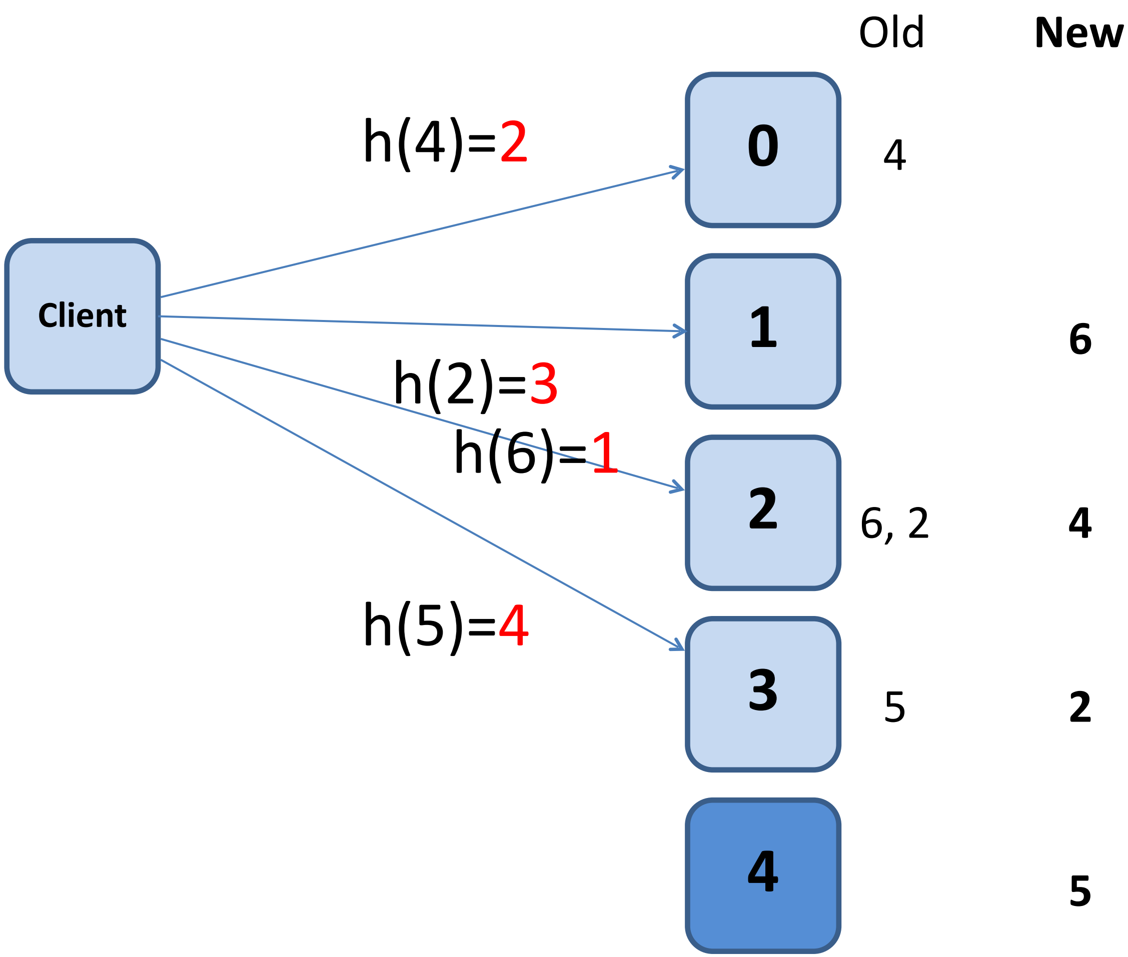
**Removing a cache changes location of almost every objects!**



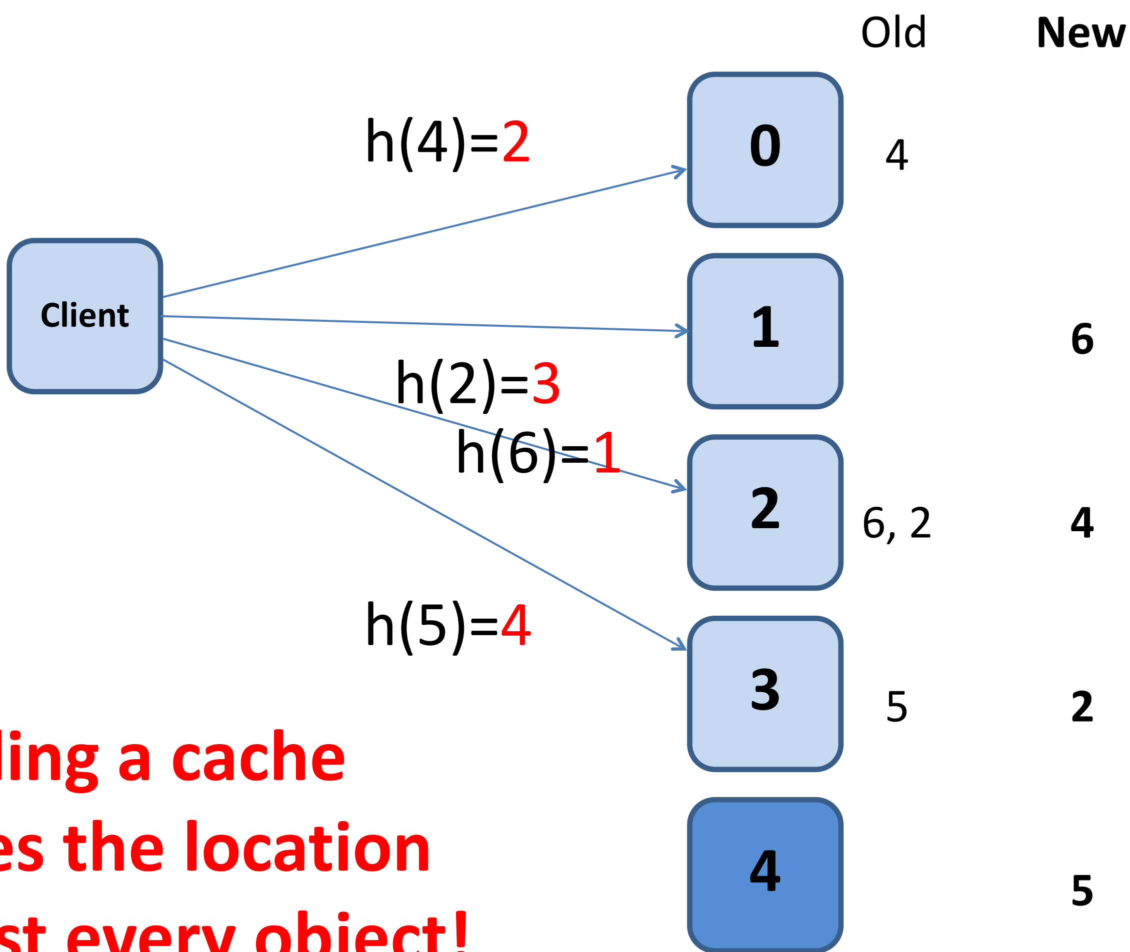
# h(u) = 7u + 4 mod 4
## (mapped across 4 nodes)



# h(u) = (7u + 4) mod 5
## (adding a cache again)



# $h(u) = (7u + 4) \bmod 5$
## (adding a cache again)




**Adding a cache changes the location of almost every object!**

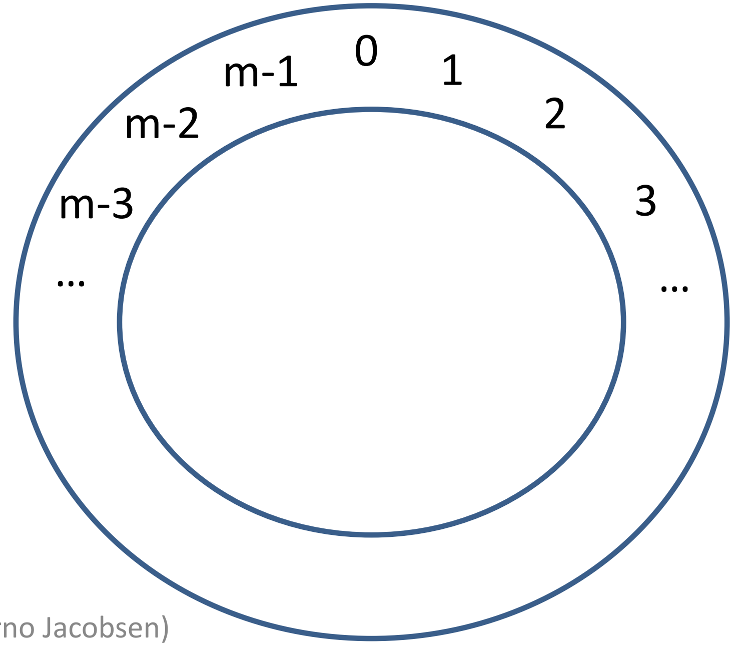
# Consistent hashing

- **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 –** remap only *n*/*m* objects (*n* objects, *m* slots)
- D. Karger *et al*., MIT, 1997
- Basis for Akamai
  - CDN company (content delivery network)
  - Web cache as a service

# Consistent hashing

## Key idea intuition

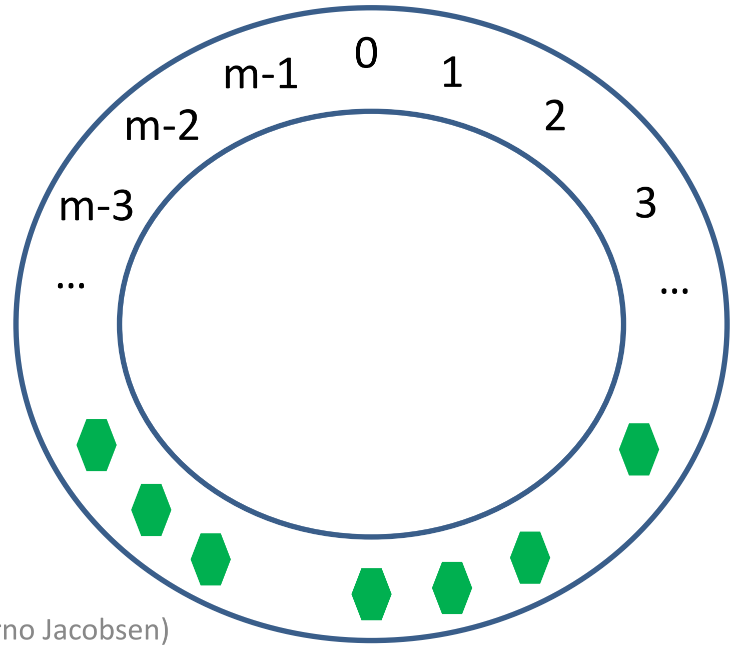
- Select a **base hash function** that maps input identifier to the number range [0, …, $m$-1]
- *E.g.*, $\boldsymbol{h(x) = (ax + b) \bmod m}$
- **Interpret range of *h*(..) as array that wraps around (i.e., a circle)**
- *h*(..) gives slot in array (circle) and wraps around at $m$-1 to 0
- Each **object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

## Key idea intuition

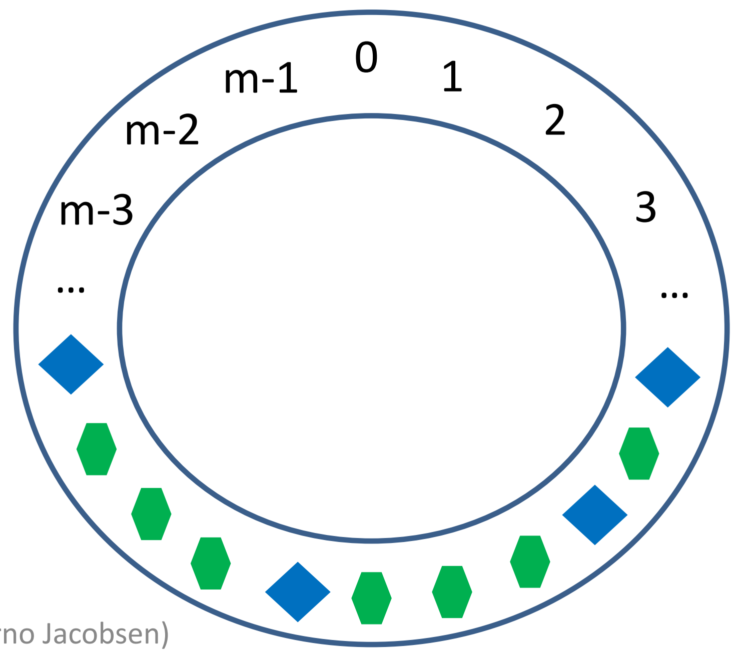
- Select a **base hash function** that maps input identifier to the number range [0, …, $m$-1]
- *E.g.*, $\boldsymbol{h(x) = (ax + b) \bmod m}$
- **Interpret range of *h*(..) as array that wraps around (i.e., a circle)**
- *h*(..) gives slot in array (circle) and wraps around at $m$-1 to 0
- Each **object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

## Key idea intuition

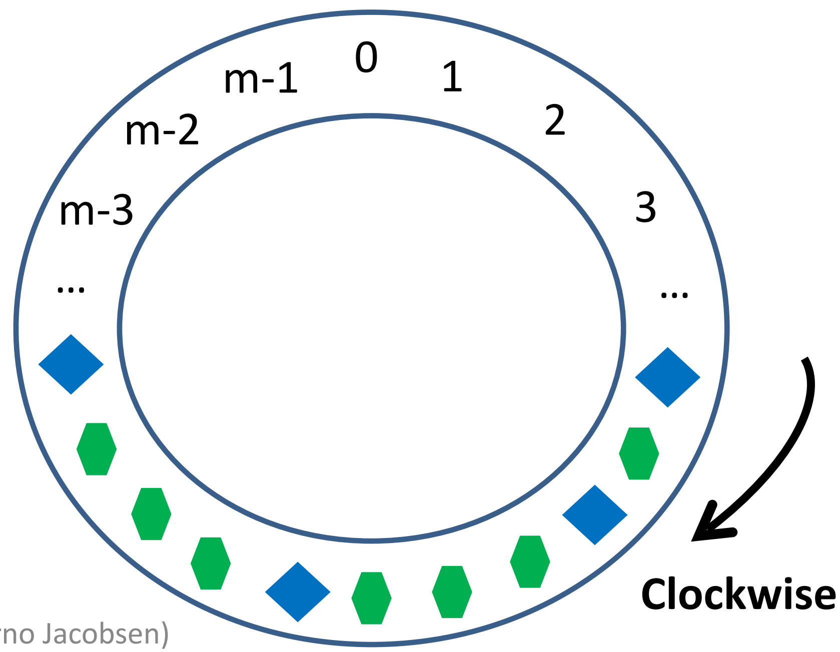
- Select a **base hash function** that maps input identifier to the number range [0, …, *m*-1]
- *E.g.*, $\mathbf{h(x) = (ax + b) \bmod m}$
- **Interpret range of *h*(..) as array that wraps around (i.e., a circle)**
- *h*(..) gives slot in array (circle) and wraps around at *m*-1 to 0
- Each **object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

## Key idea intuition

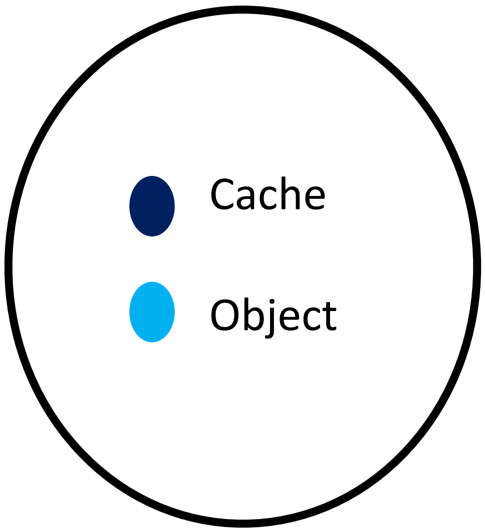
- Select a **base hash function** that maps input identifier to the number range [0, …, $m$-1]
- *E.g.*, $\boldsymbol{h(x) = (ax + b) \bmod m}$
- **Interpret range of *h*(..) as array that wraps around (i.e., a circle)**
- *h*(..) gives slot in array (circle) and wraps around at $m$-1 to 0
- Each **object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

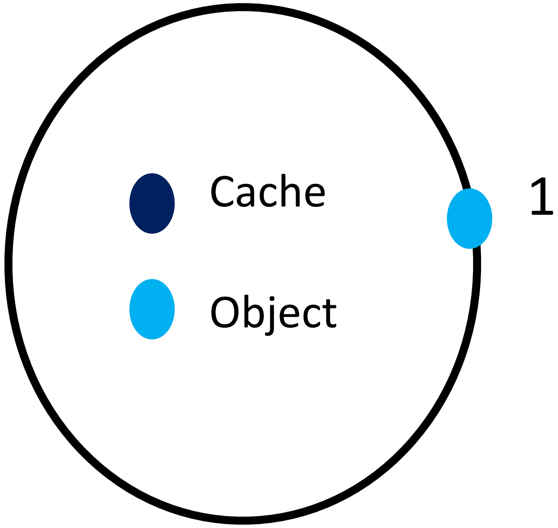
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

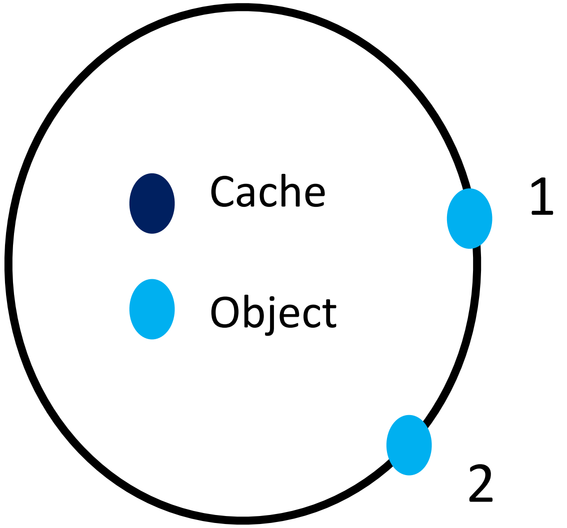
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

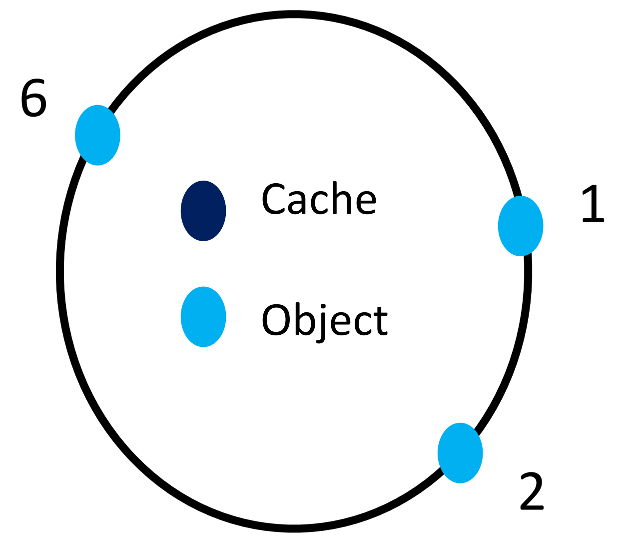
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

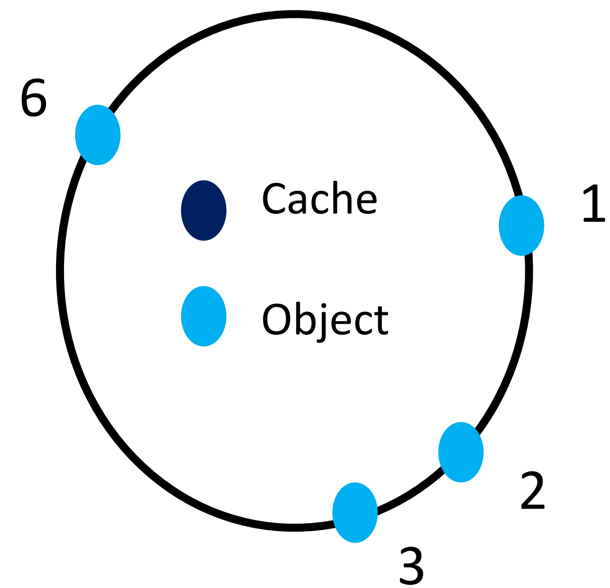
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

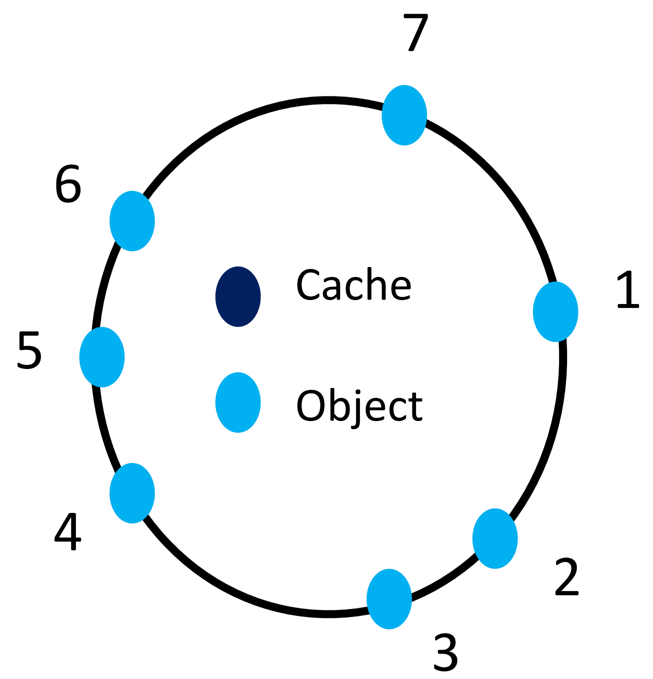
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

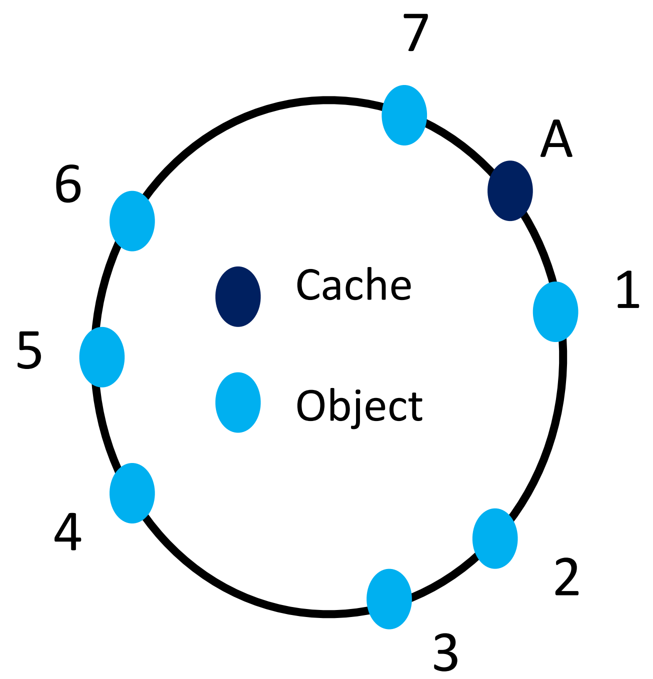
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

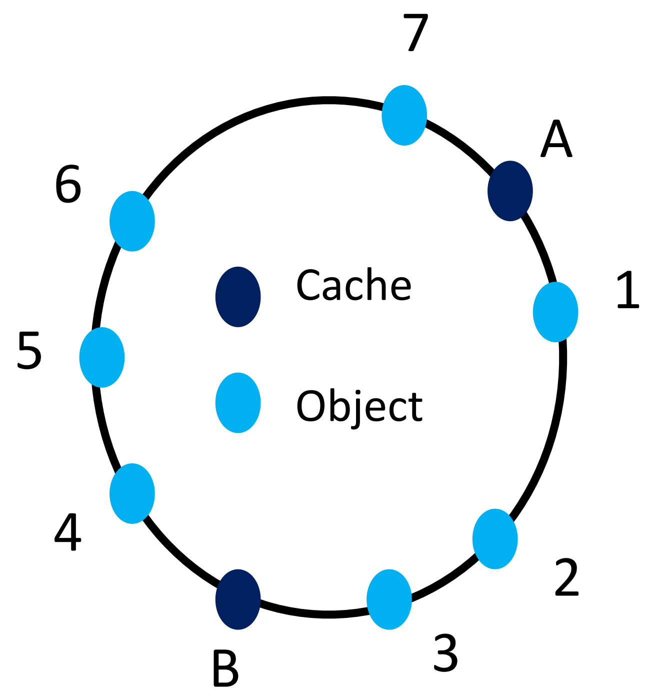
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

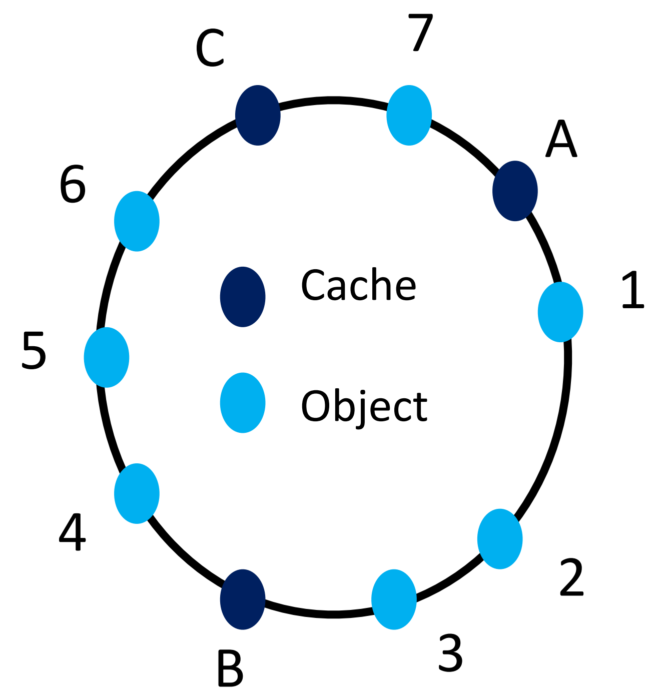
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



# Consistent hashing

## Original interpretation

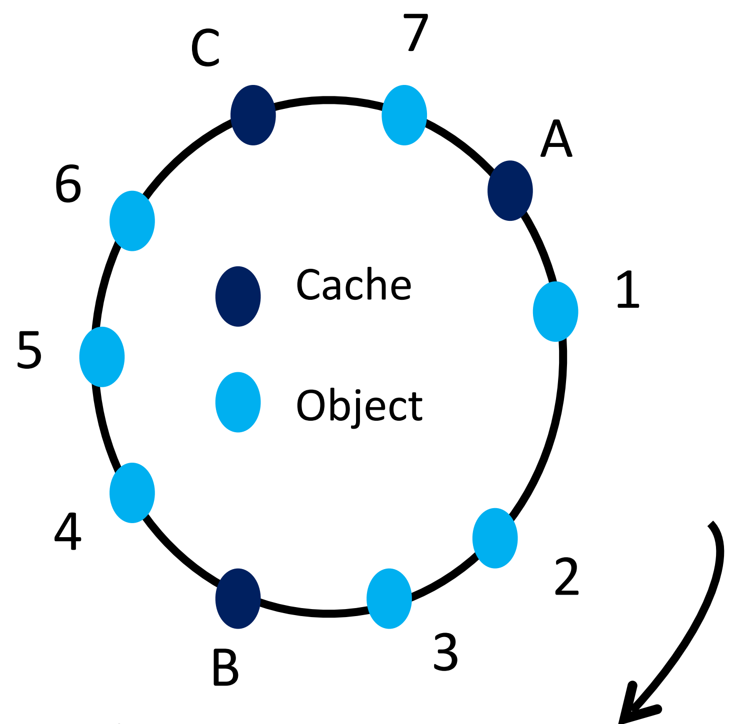
- Select a **base hash function** that maps input identifier to the number range [0, …, M]
- Divide by M, re-mapping **[0,…,M]** to **[0, 1]**
- Interpret this interval as the **unit circle***:* Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on unit circle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



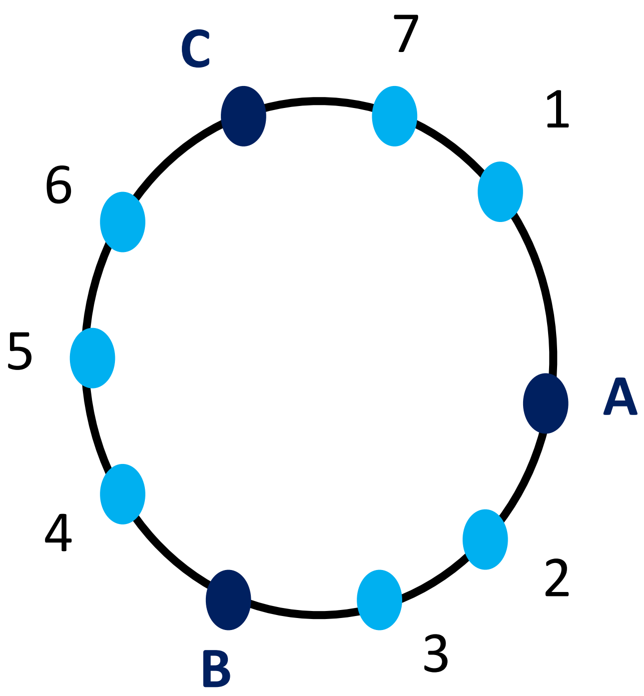
# Consistent hashing

## 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

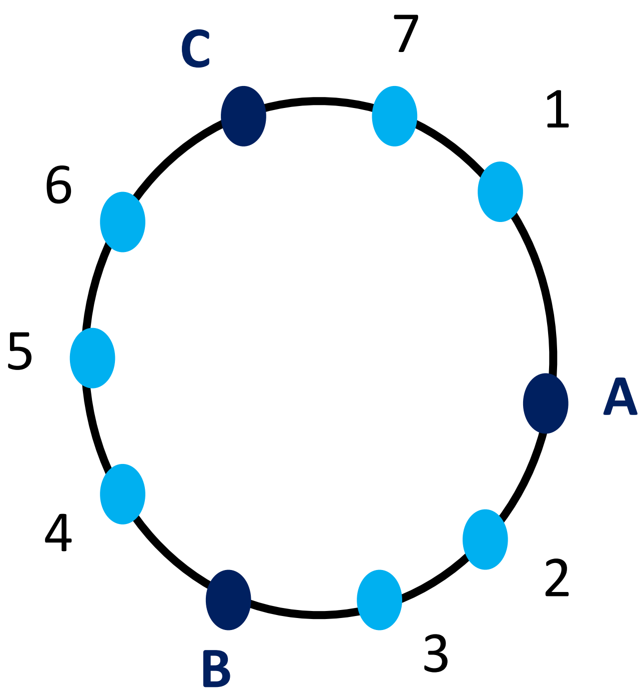



| | | |
|---|---|---|
| Items | 2, 3 | mapped to **B** |
| Items | 4, 5, 6 | mapped to **C** |
| Items | 7, 1 | mapped to **A** |

Cache

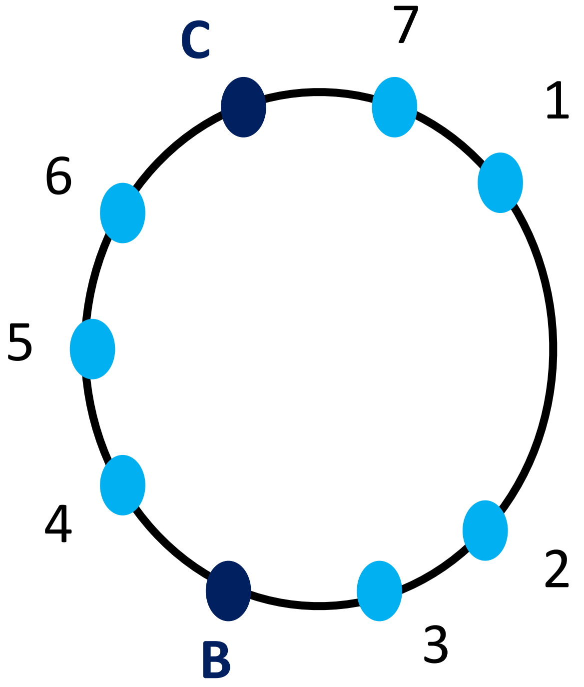
Object

# Removing a cache



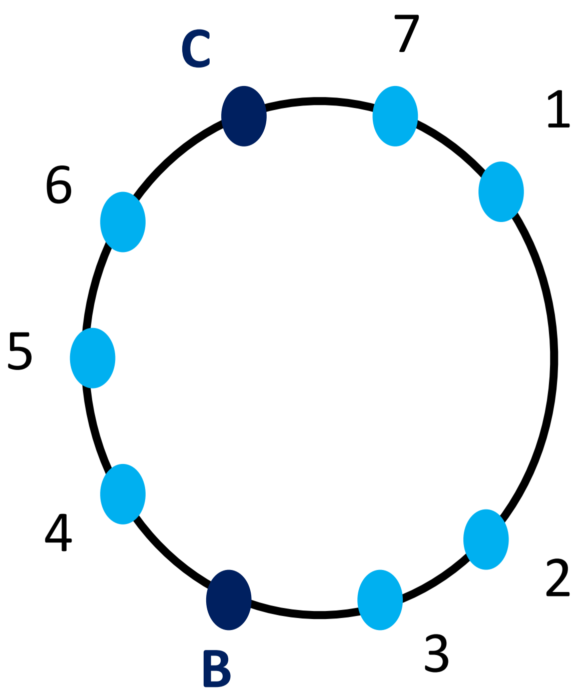

| | | |
|---|---|---|
| Items | 2, 3 | mapped to **B** |
| Items | 4, 5, 6 | mapped to **C** |
| Items | 7, 1 | mapped to **A** |

# Removing a cache




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

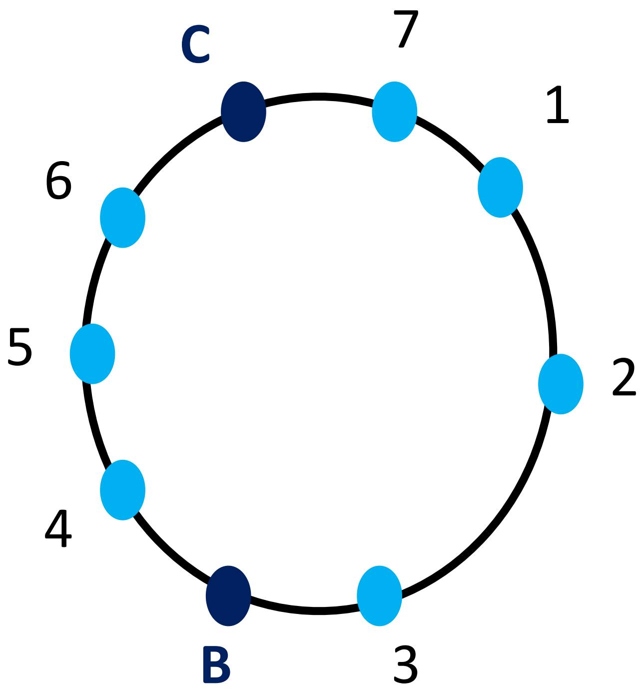
# Removing a cache




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

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

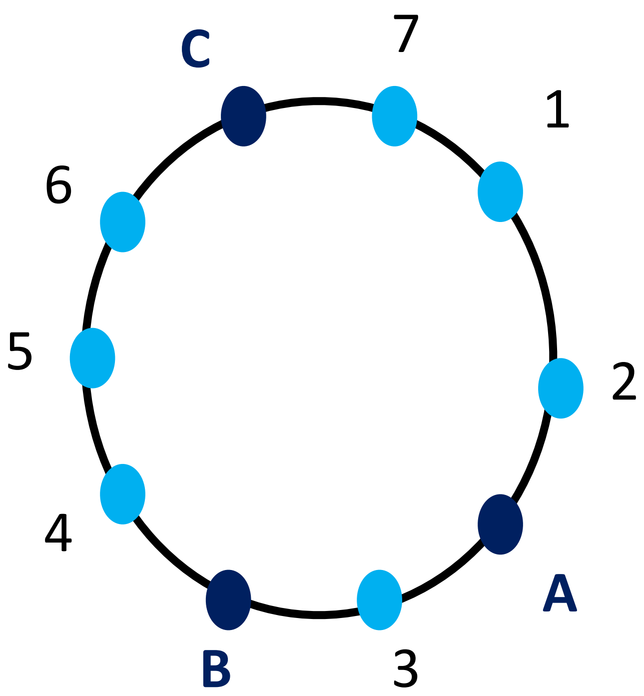
# Adding a cache




Items 7, 1, 2, 3 mapped to B

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

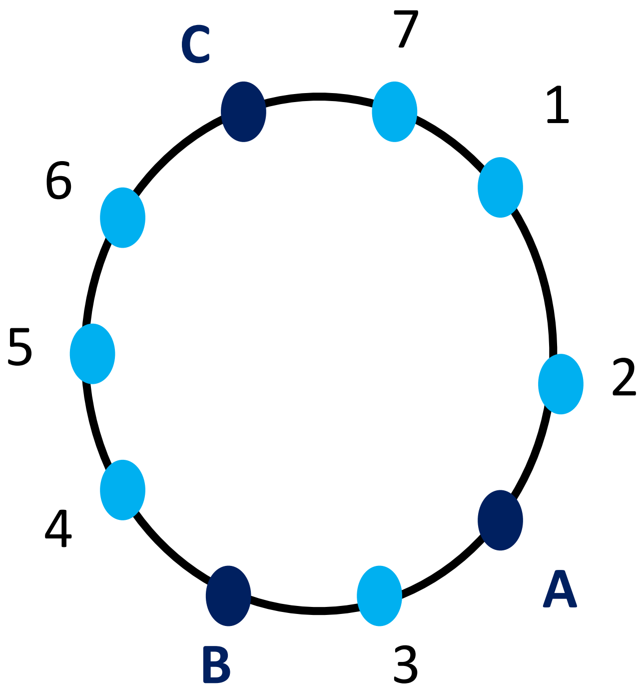
# Adding a cache




Items 7, 1, 2, 3 mapped to B

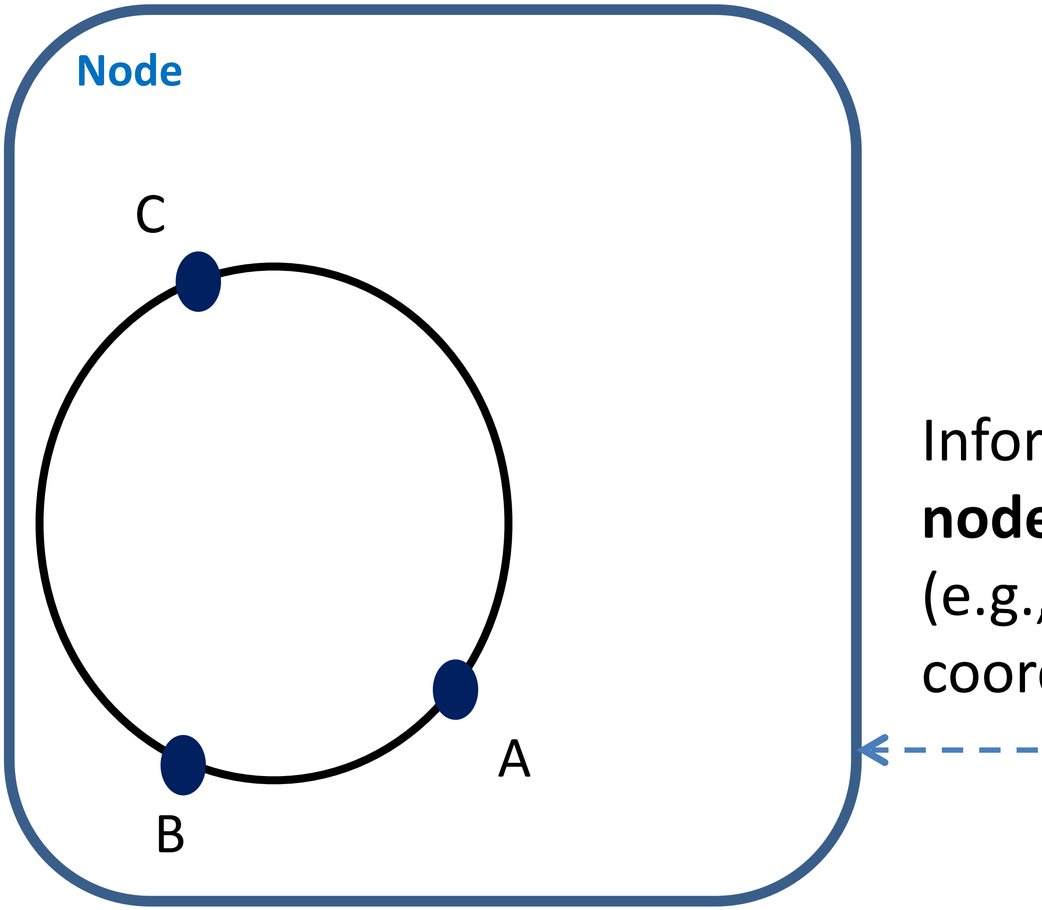
Items 4, 5, 6 mapped to **C**

# Adding a cache



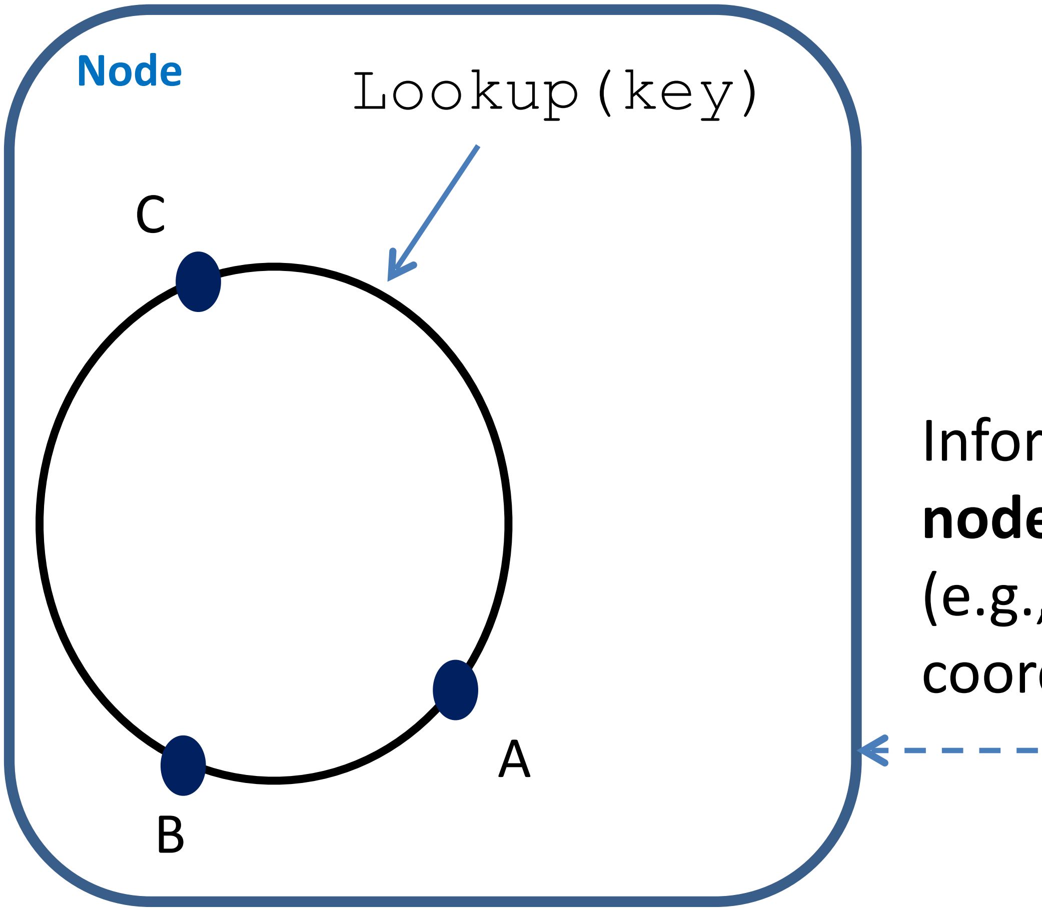

| | | |
|---|---|---|
| Items | 3 | mapped to **B** |
| Items | 4, 5, 6 | mapped to **C** |
| Items | **7, 1, 2** | mapped to **A** |

# Processing a `Lookup(key)`



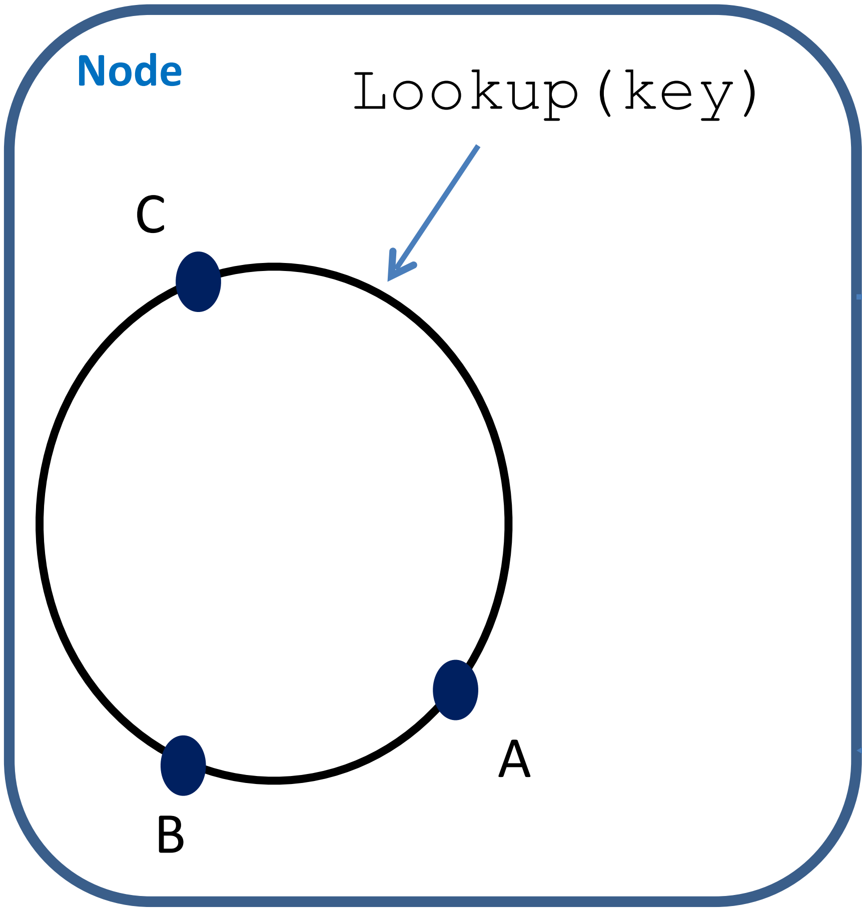

Information about **node addition** & **removal** (e.g., via gossiping or via a coordination service)

# Processing a `Lookup(key)`




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

# Processing a `Lookup(key)`



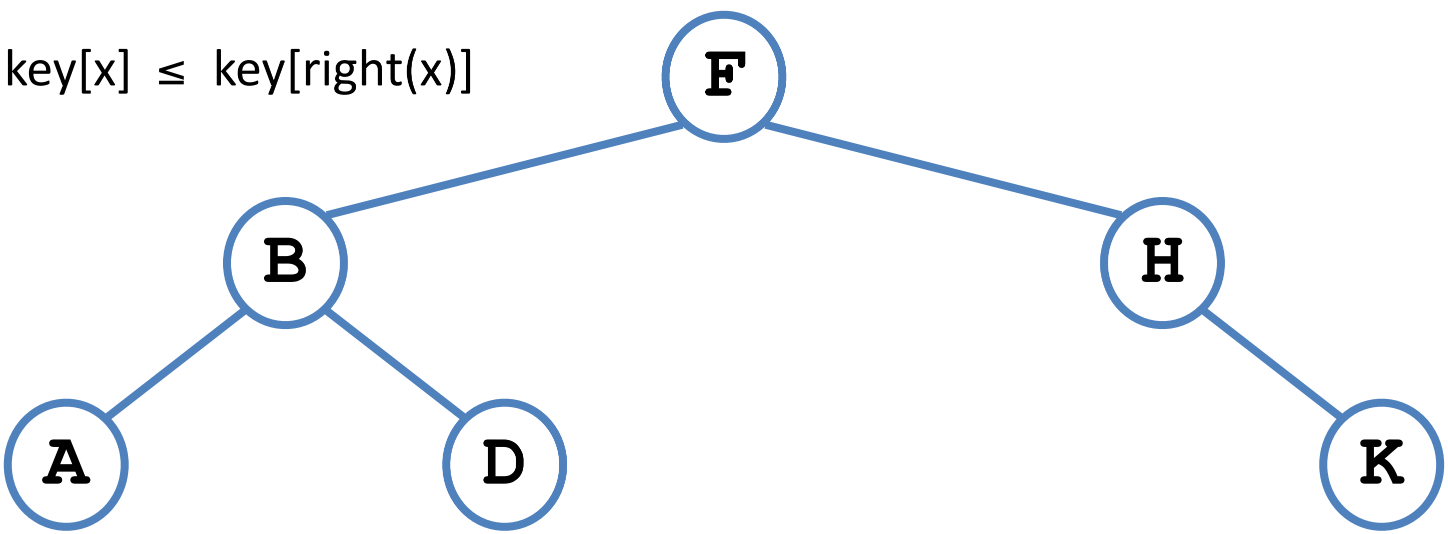

Retrieve ***object*** with ***key*** from ***A***

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)] ≤ key[x] ≤ key[right(x)]



# 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

# 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

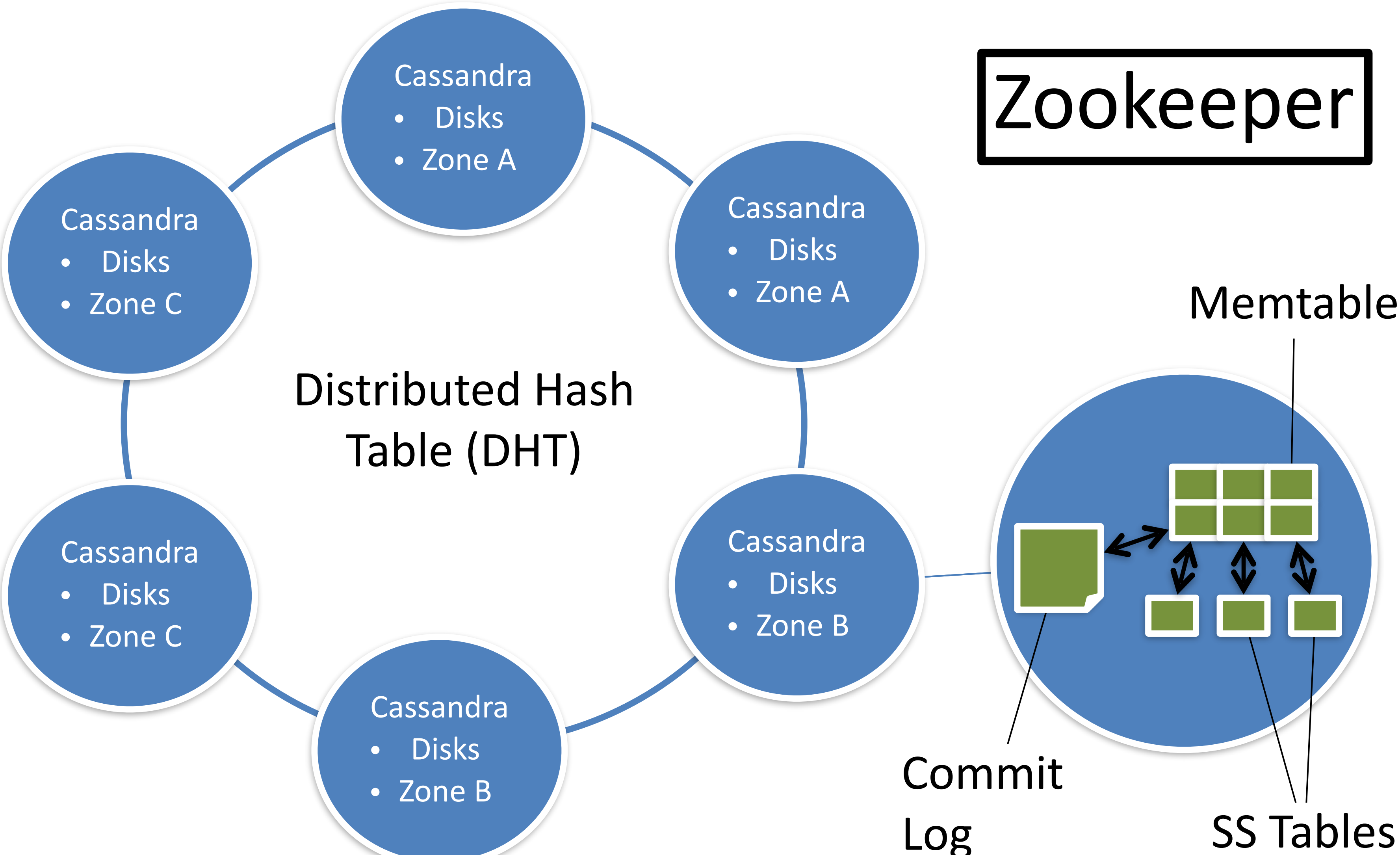
http://en.wikipedia.org/wiki/MD5#Algorithm

# DYNAMO / CASSANDRA

# Cassandra

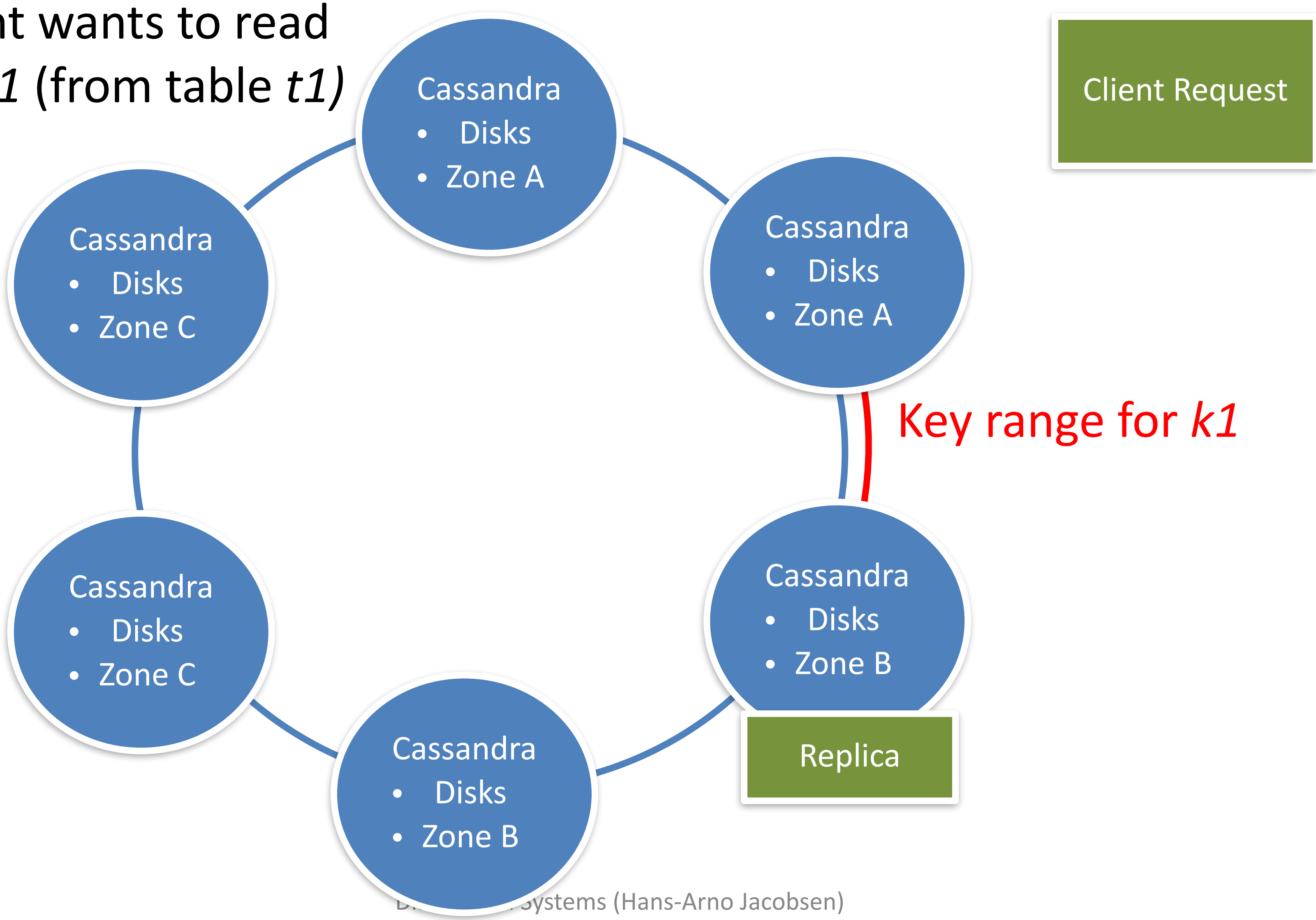
- Developed by Facebook
- Based on Amazon Dynamo (but open-source)
- Structured storage nodes (**no GFS** used)
- **Decentralized** architecture (no master assignment)
- **Consistent hashing** for load balancing
- Eventual consistency
- **Gossiping** to exchange information

# Cassandra architecture overview



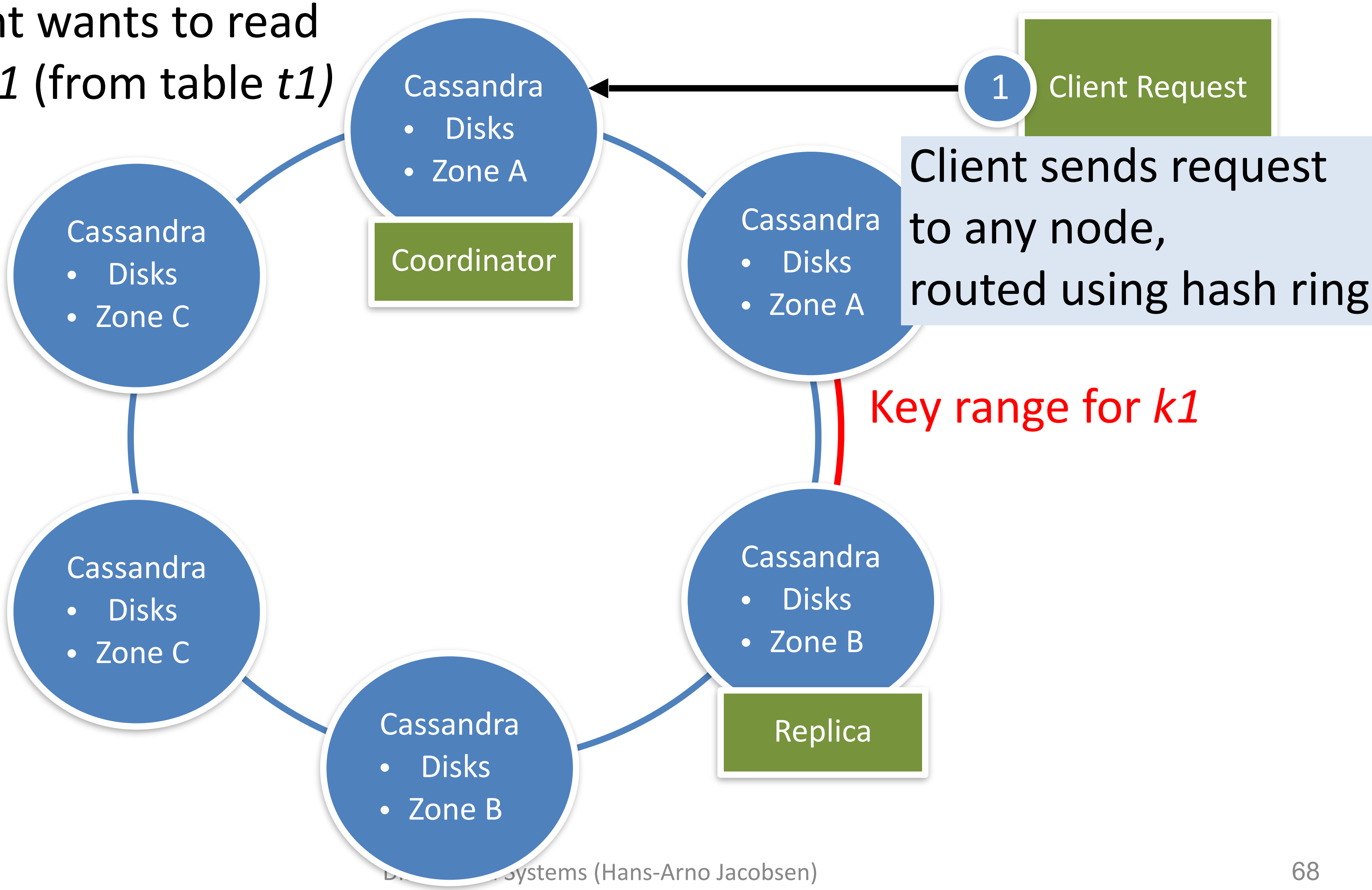
# Cassandra global read-path

Client wants to read key *k1* (from table *t1)*

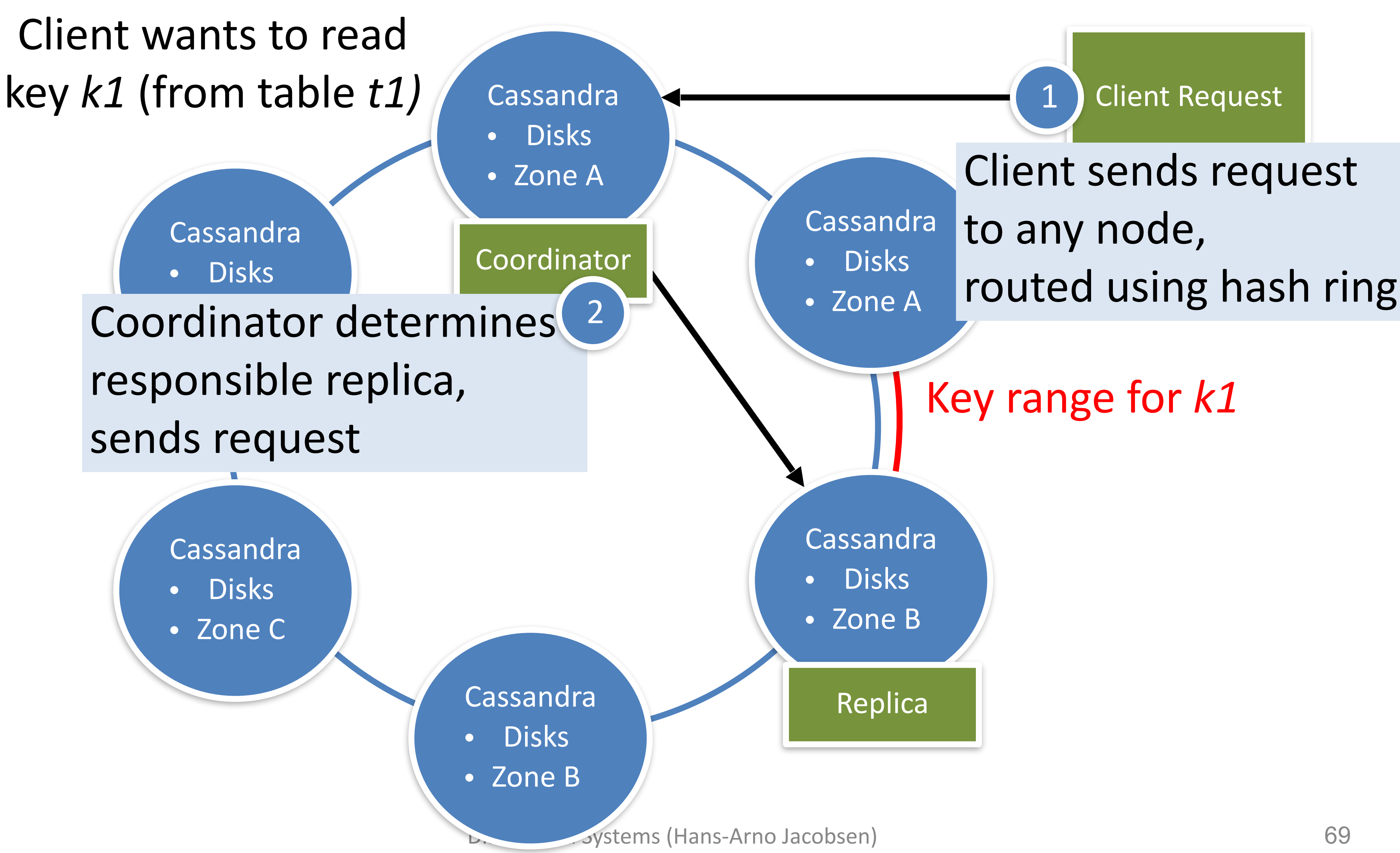


# Cassandra global read-path

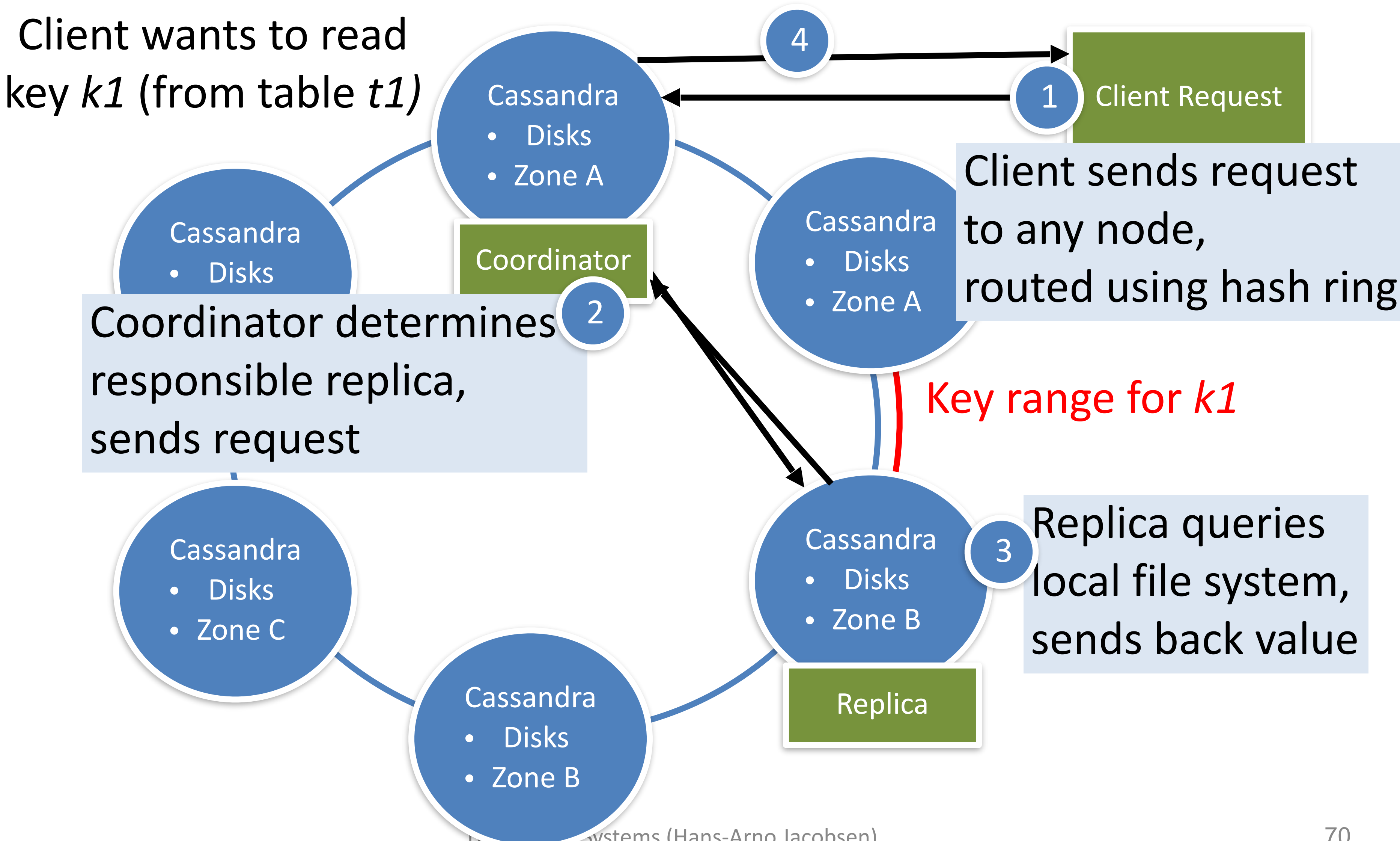
Client wants to read key *k1* (from table *t1)*



# Cassandra global read-path

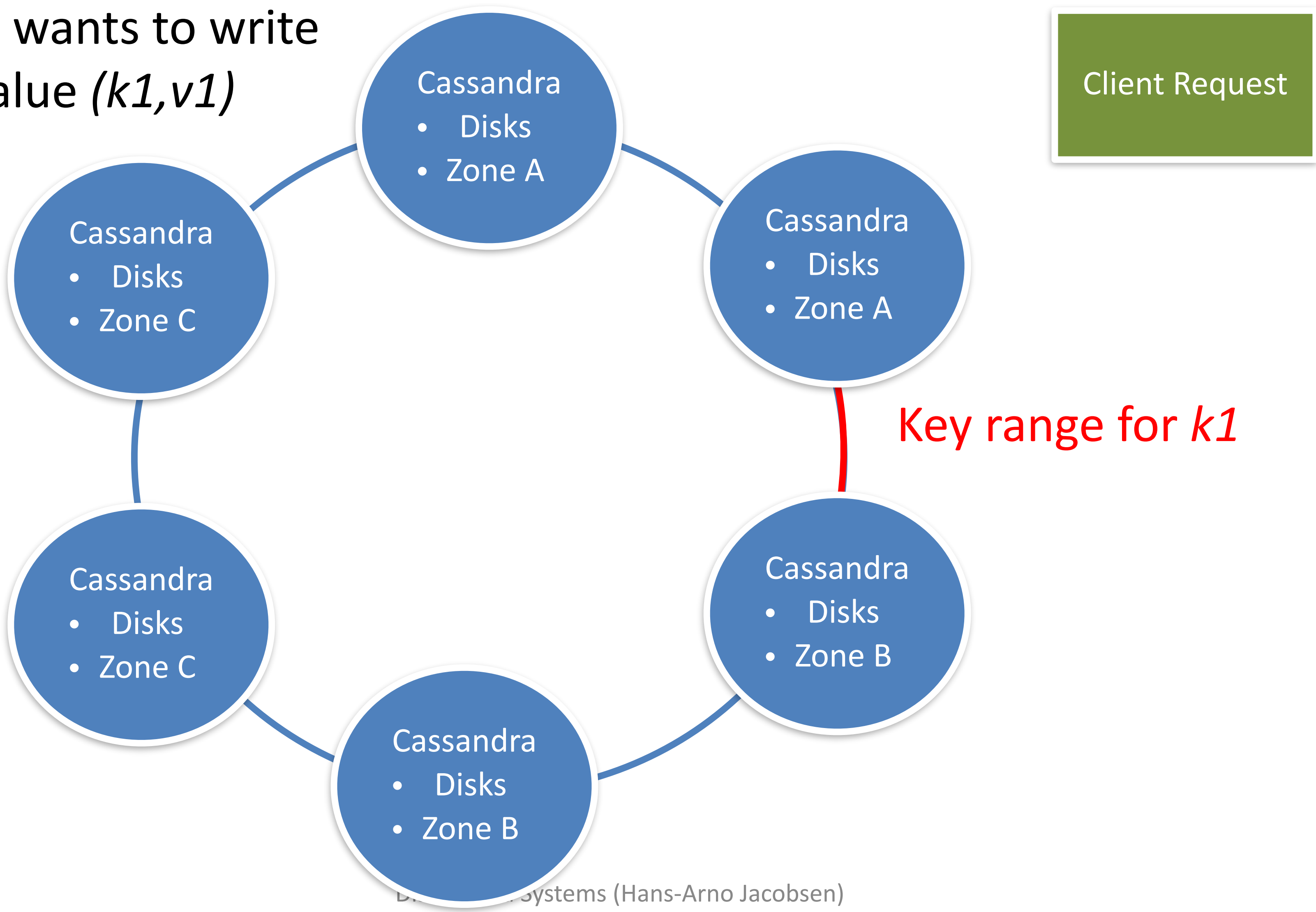


# Cassandra global read-path



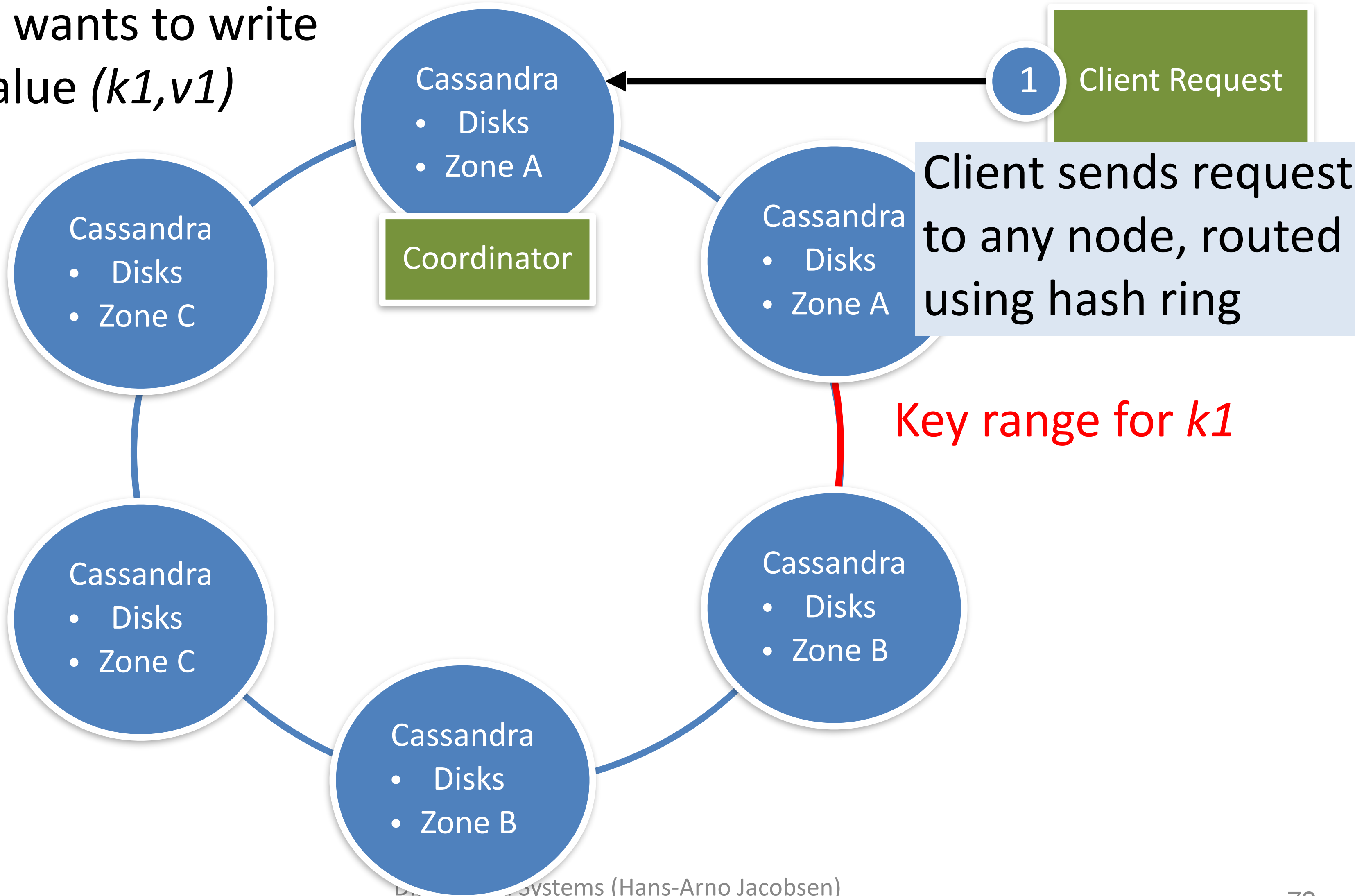
# Cassandra global write-path

Client wants to write key-value *(k1,v1)*



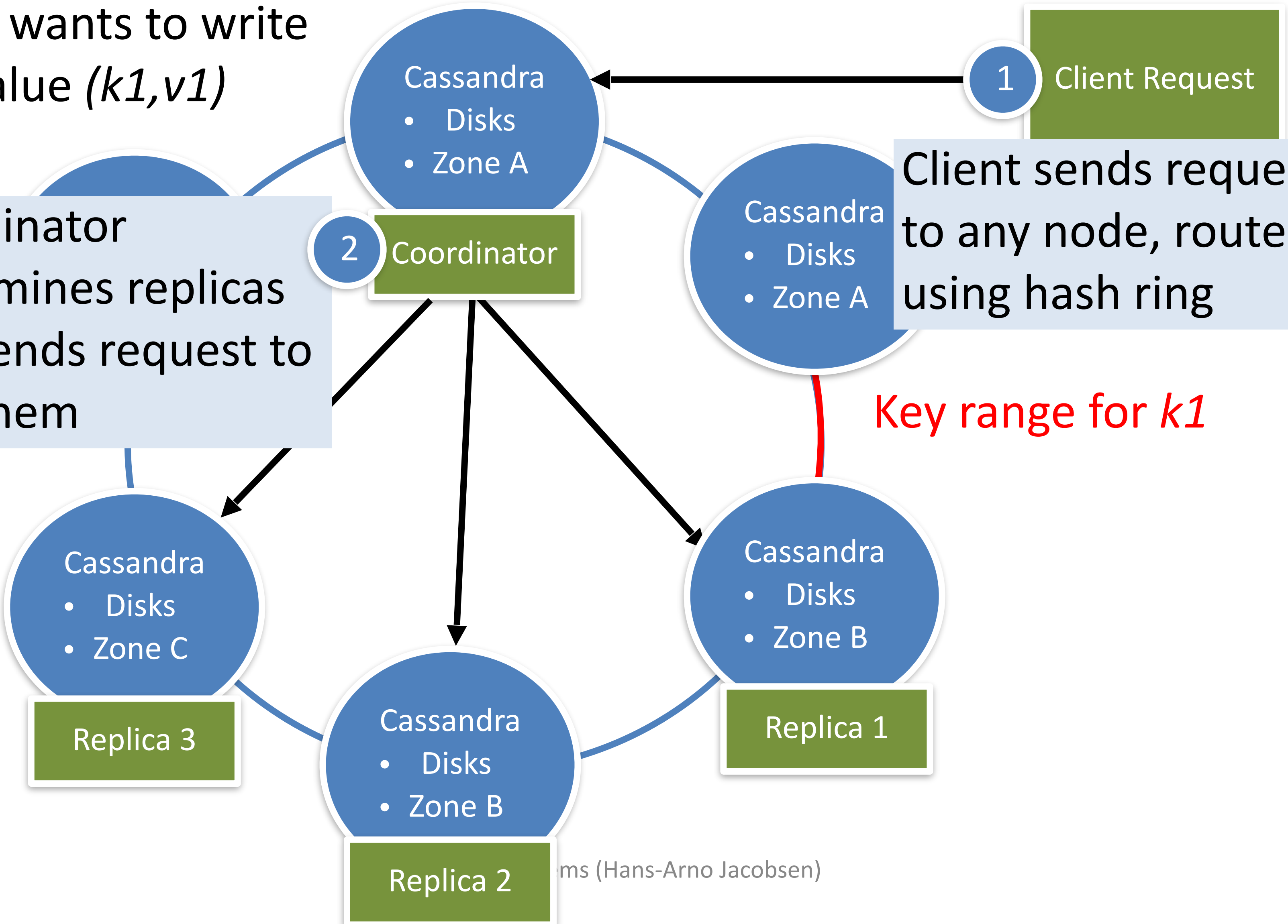
# Cassandra global write-path

Client wants to write key-value *(k1,v1)*



# Cassandra global write-path

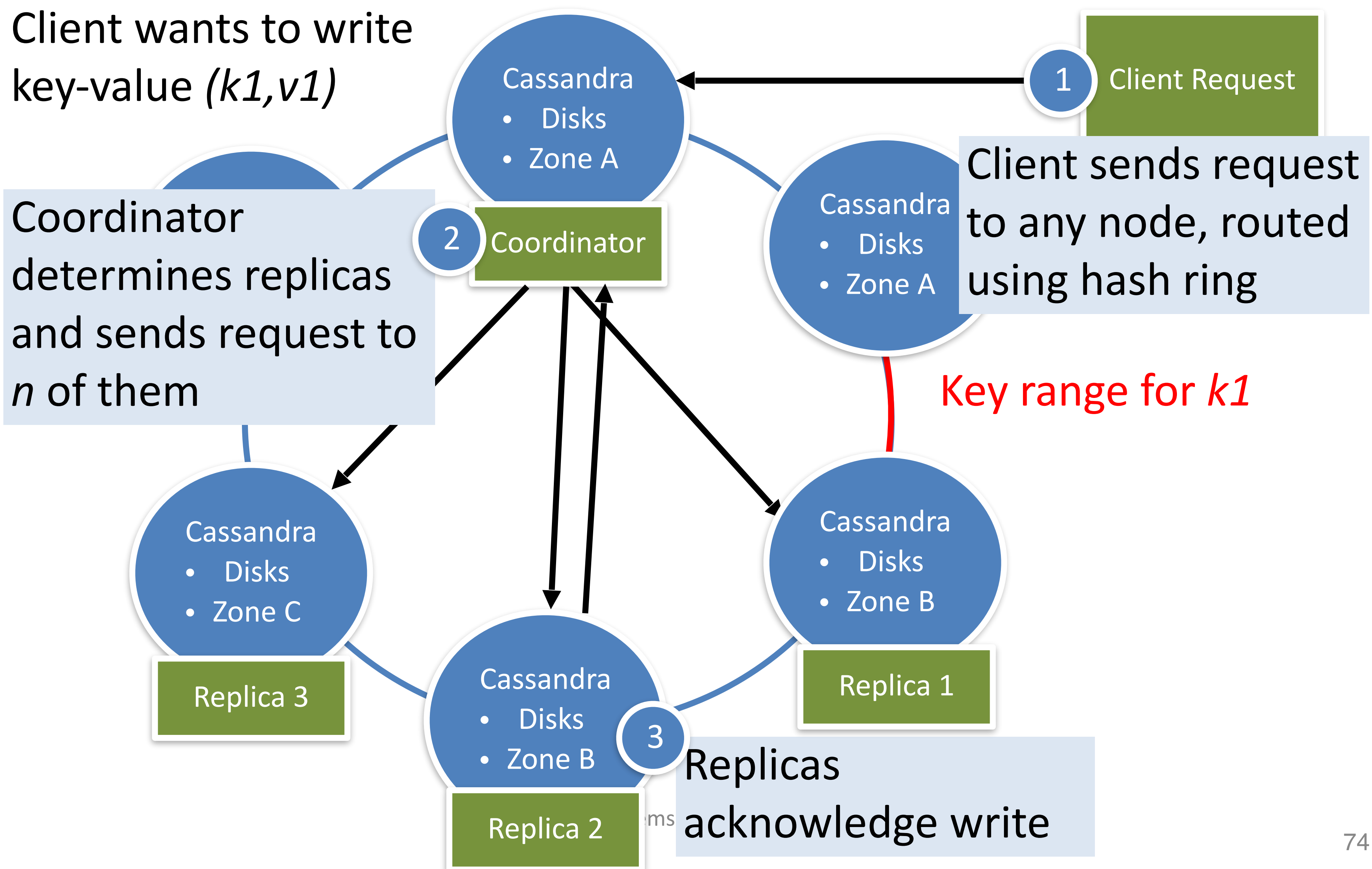
Client wants to write key-value *(k1,v1)*



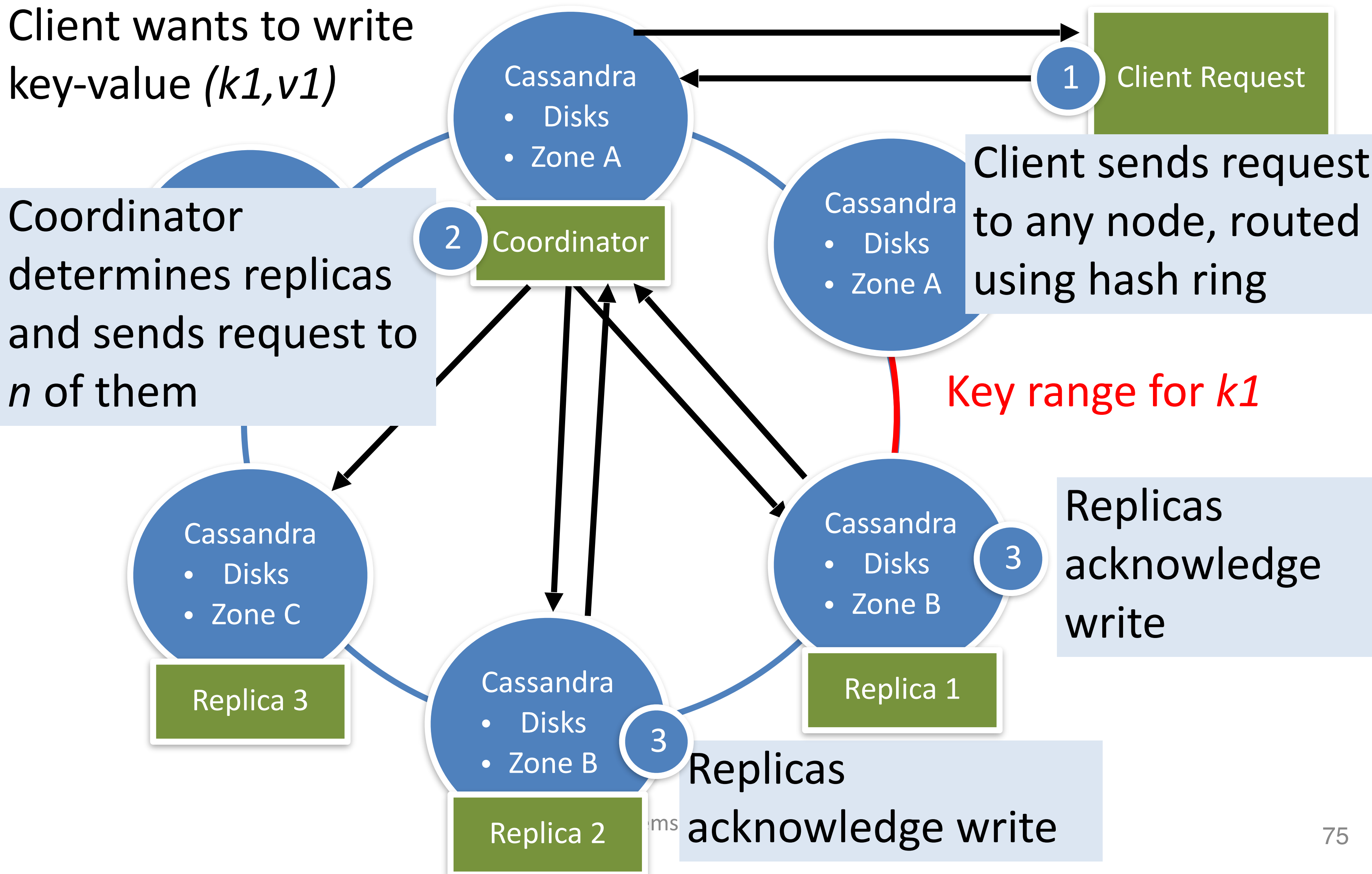

Client sends request to any node, routed using hash ring

Coordinator determines replicas and sends request to *n* of them

# Cassandra global write-path

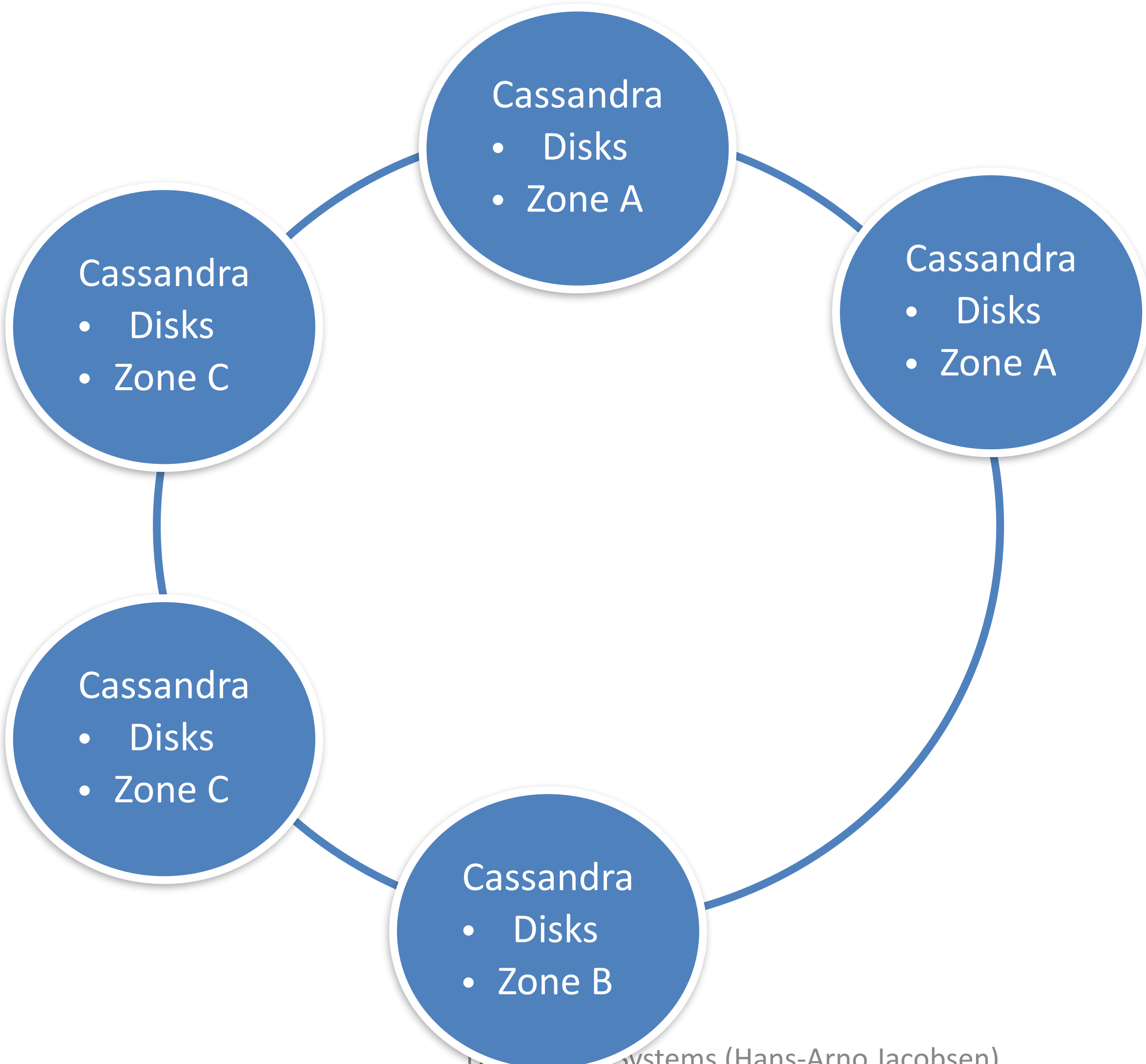


# Cassandra global write-path



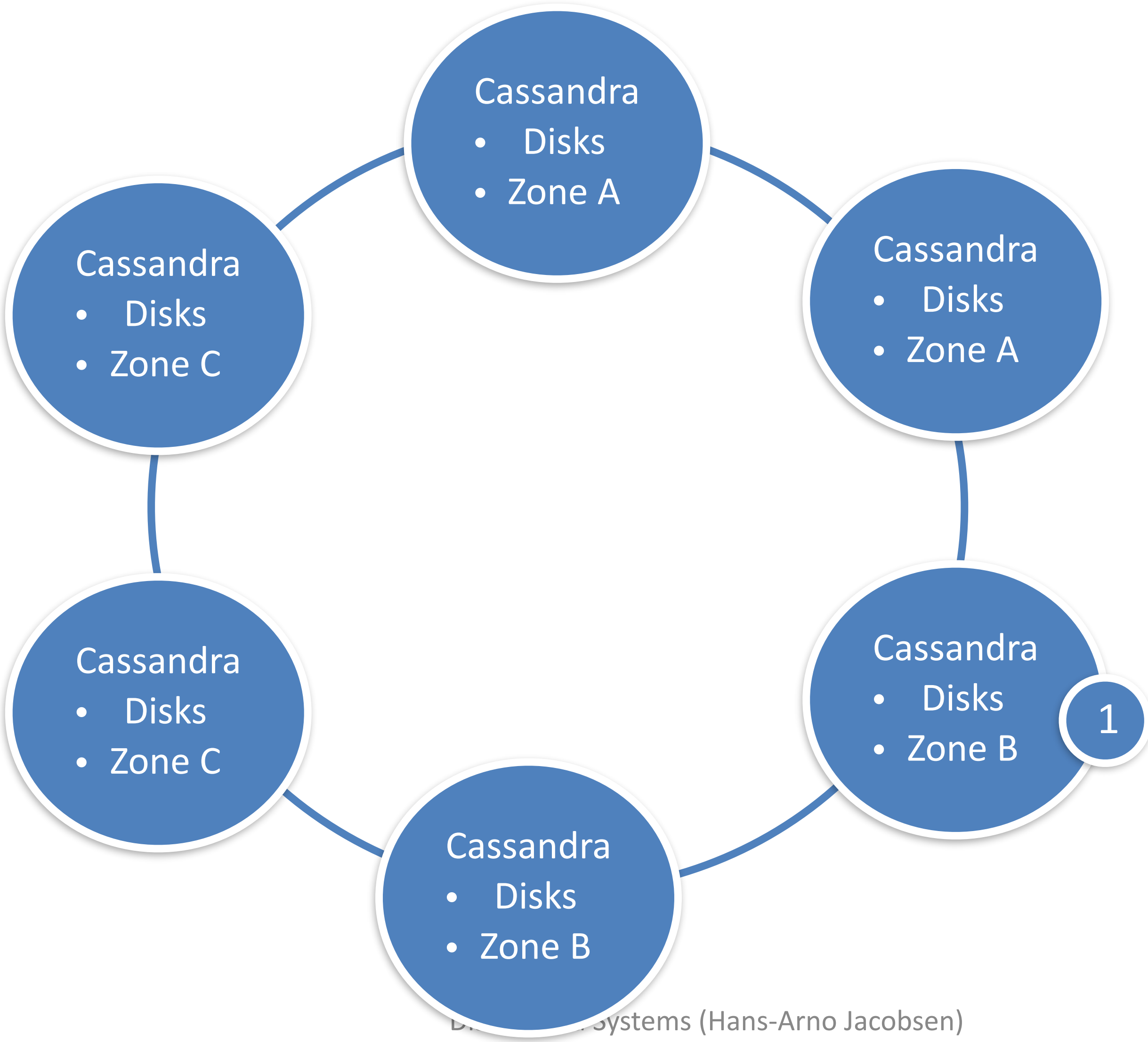
# Incremental scaling in Cassandra

## (i.e., adding a storage unit)



# Incremental scaling in Cassandra

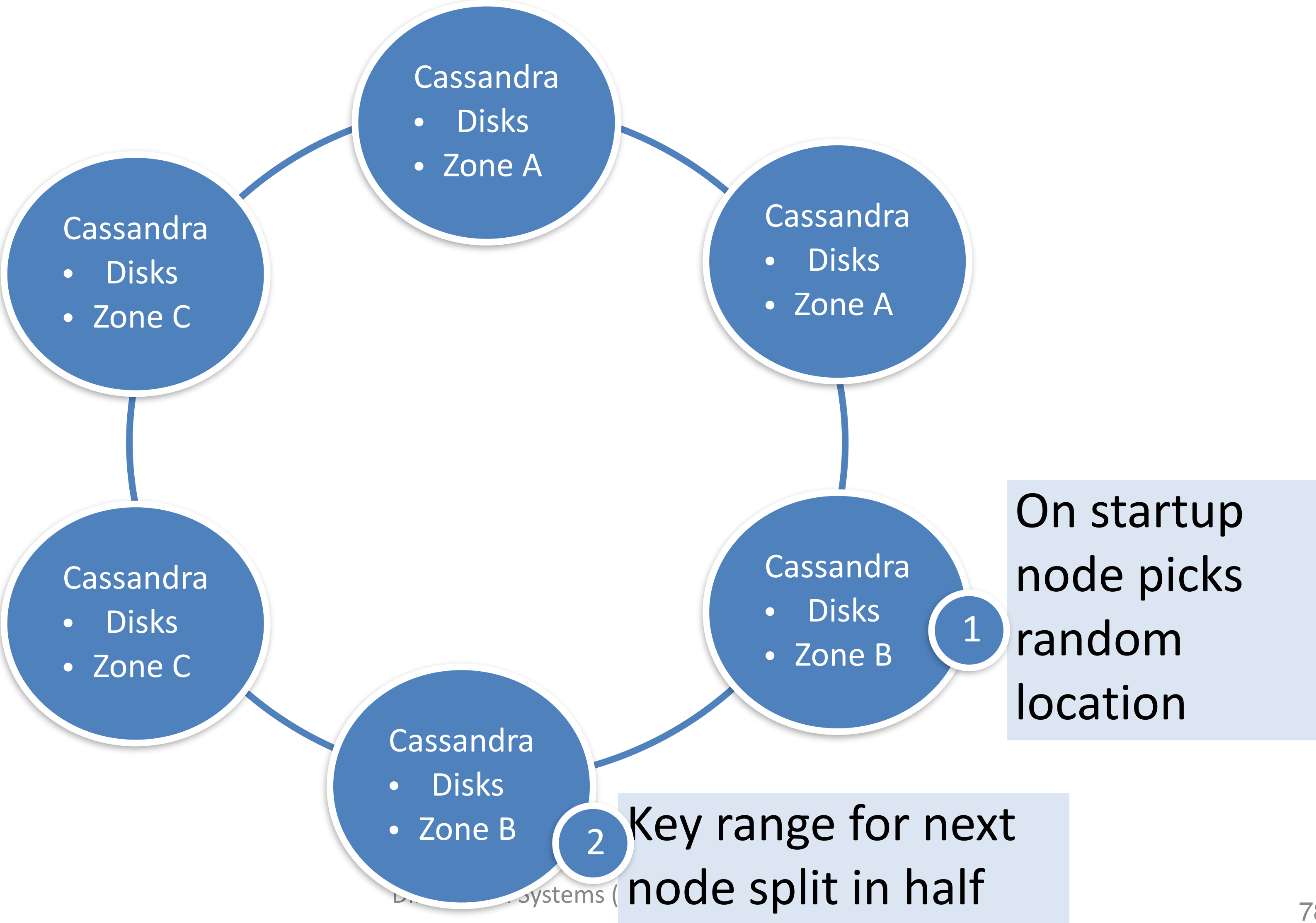
## (i.e., adding a storage unit)




On startup node picks random location

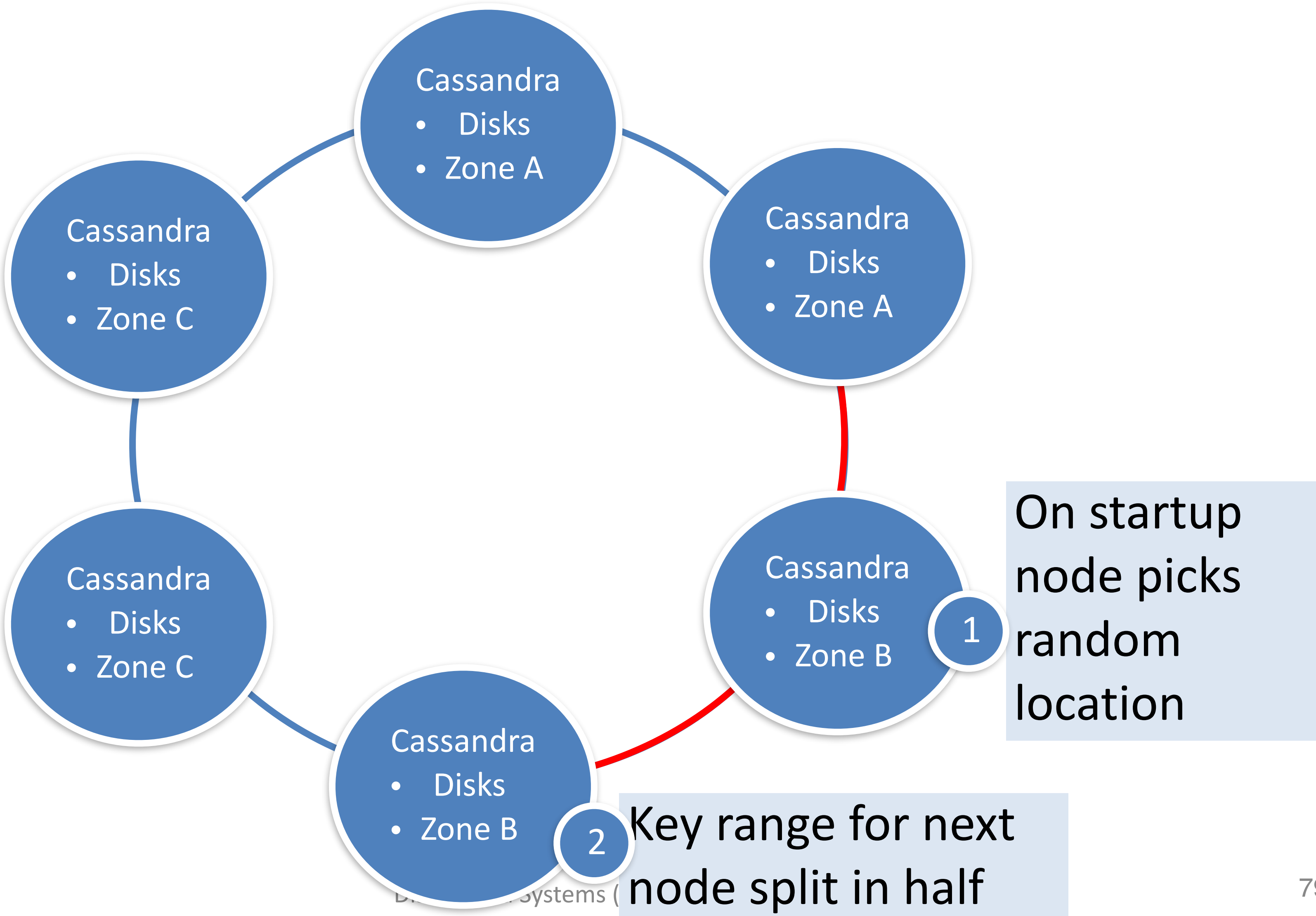
# Incremental scaling in Cassandra

## (i.e., adding a storage unit)



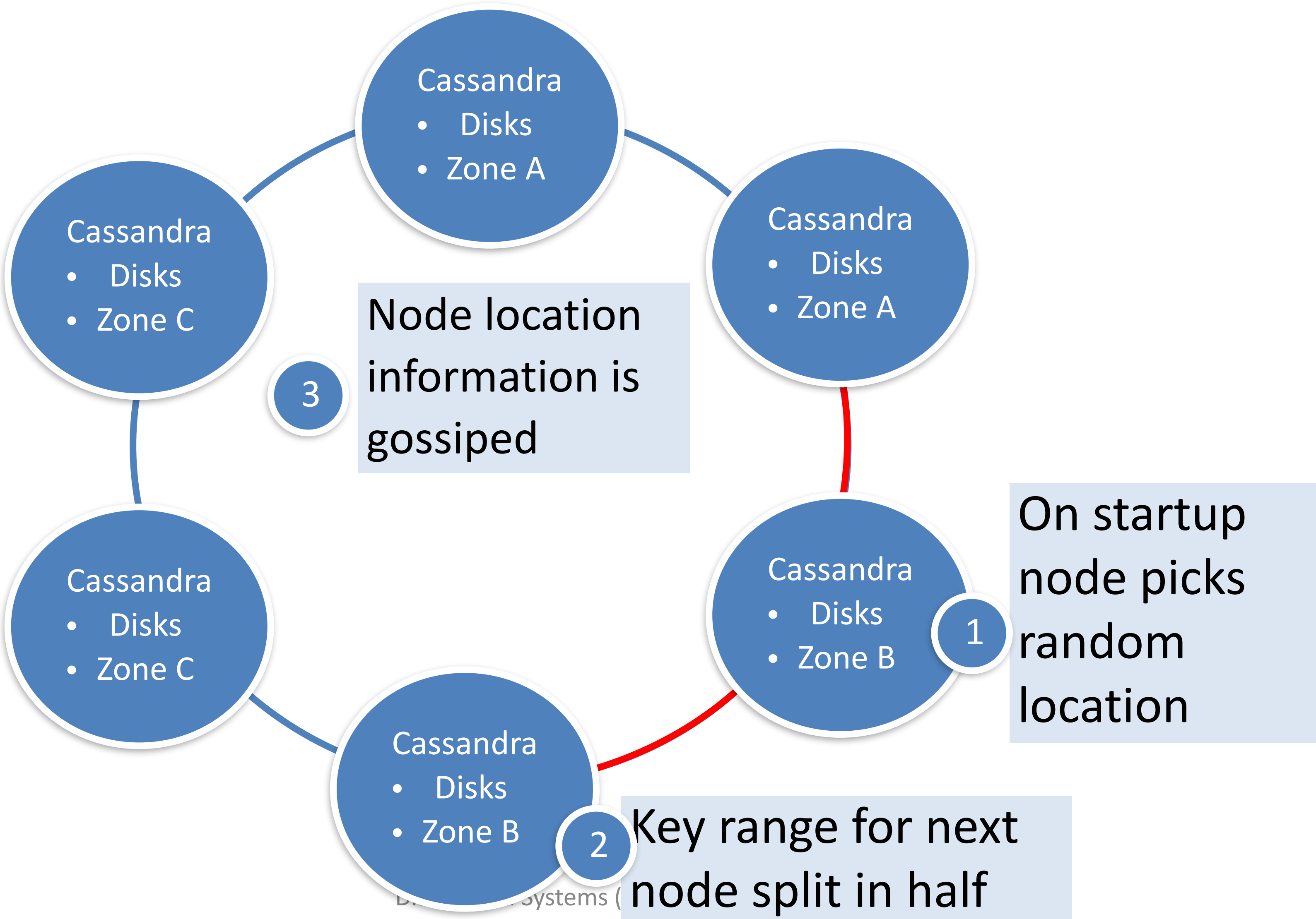
# Incremental scaling in Cassandra

## (i.e., adding a storage unit)



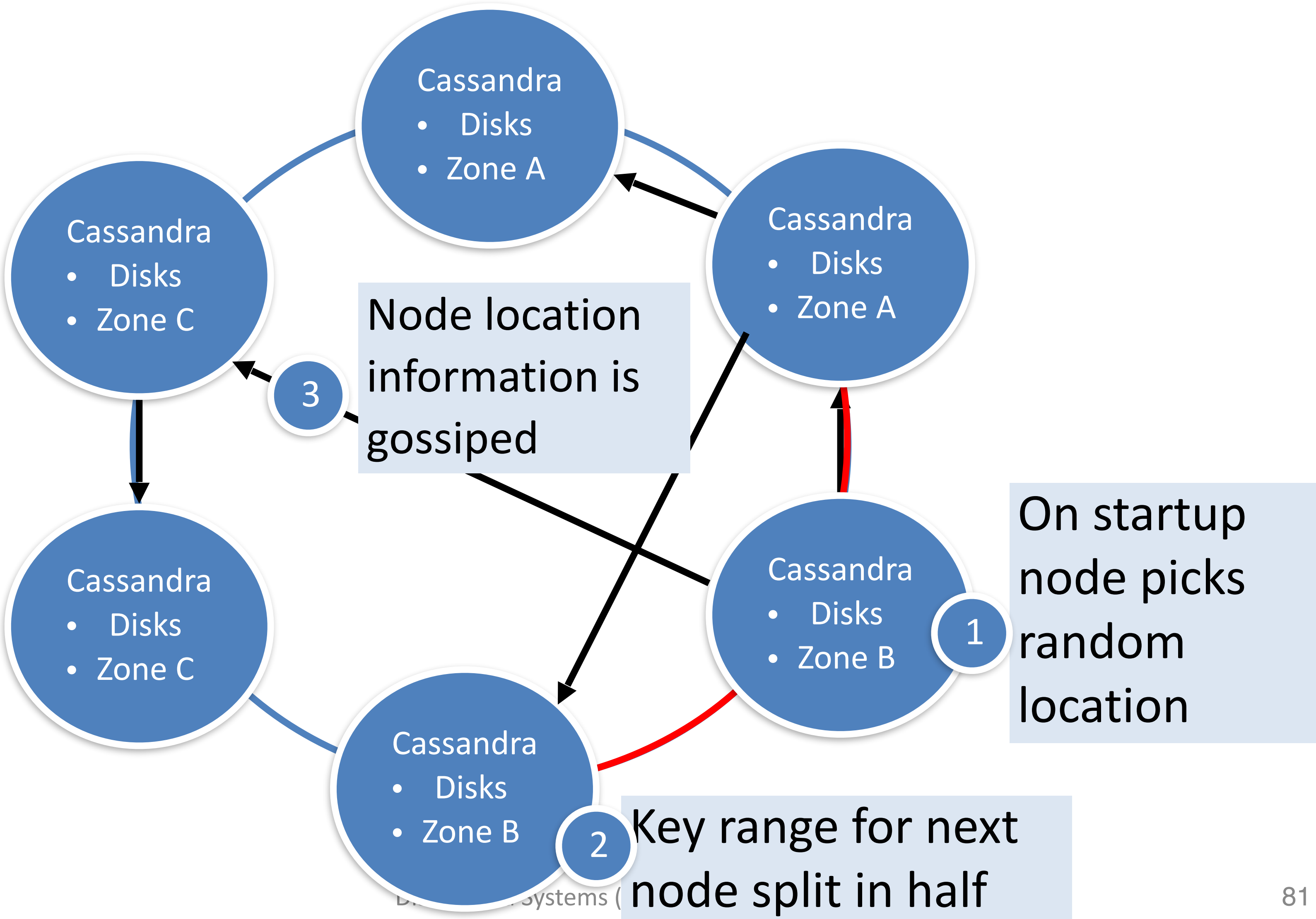
# Incremental scaling in Cassandra

## (i.e., adding a storage unit)

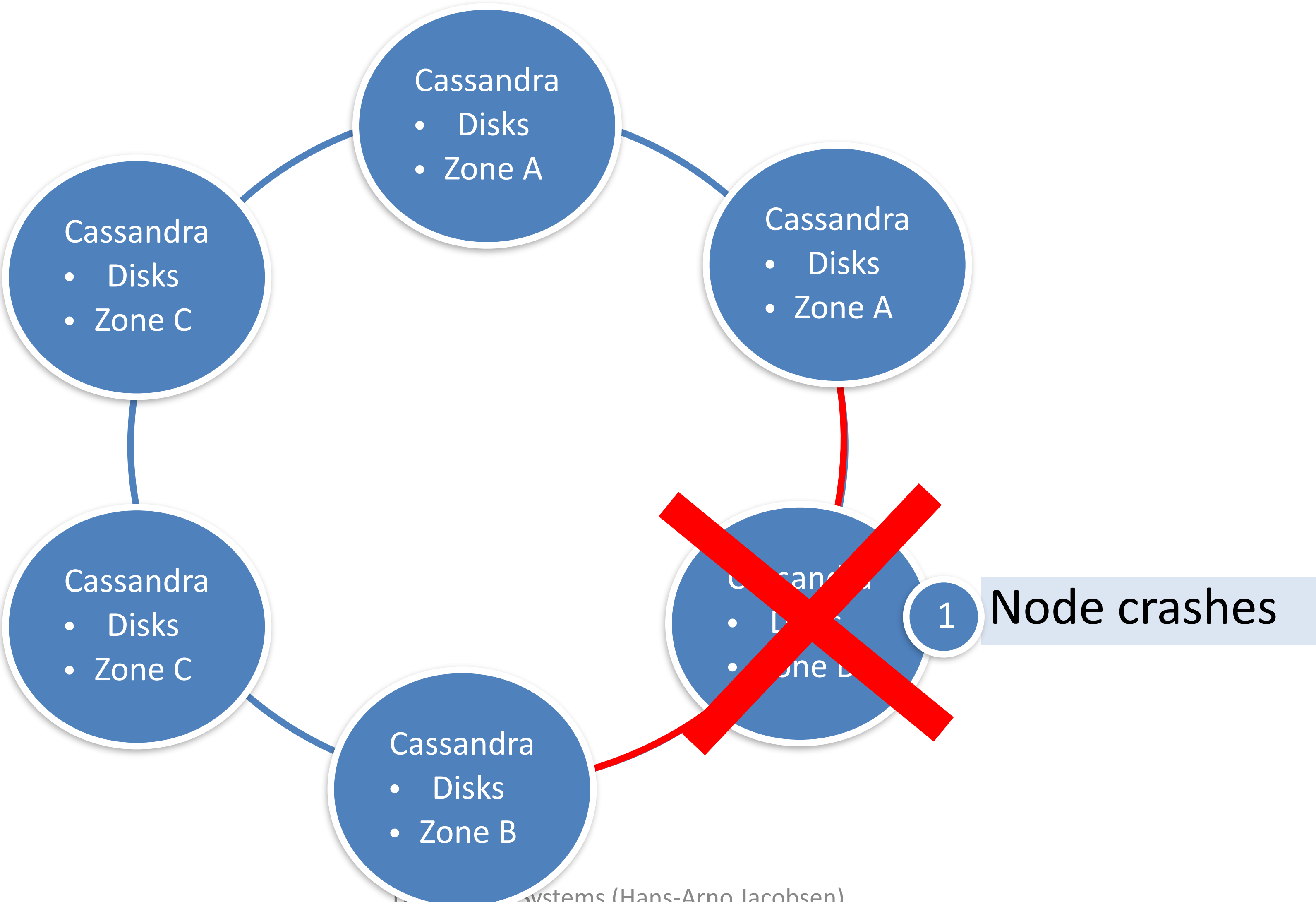


# Incremental scaling in Cassandra

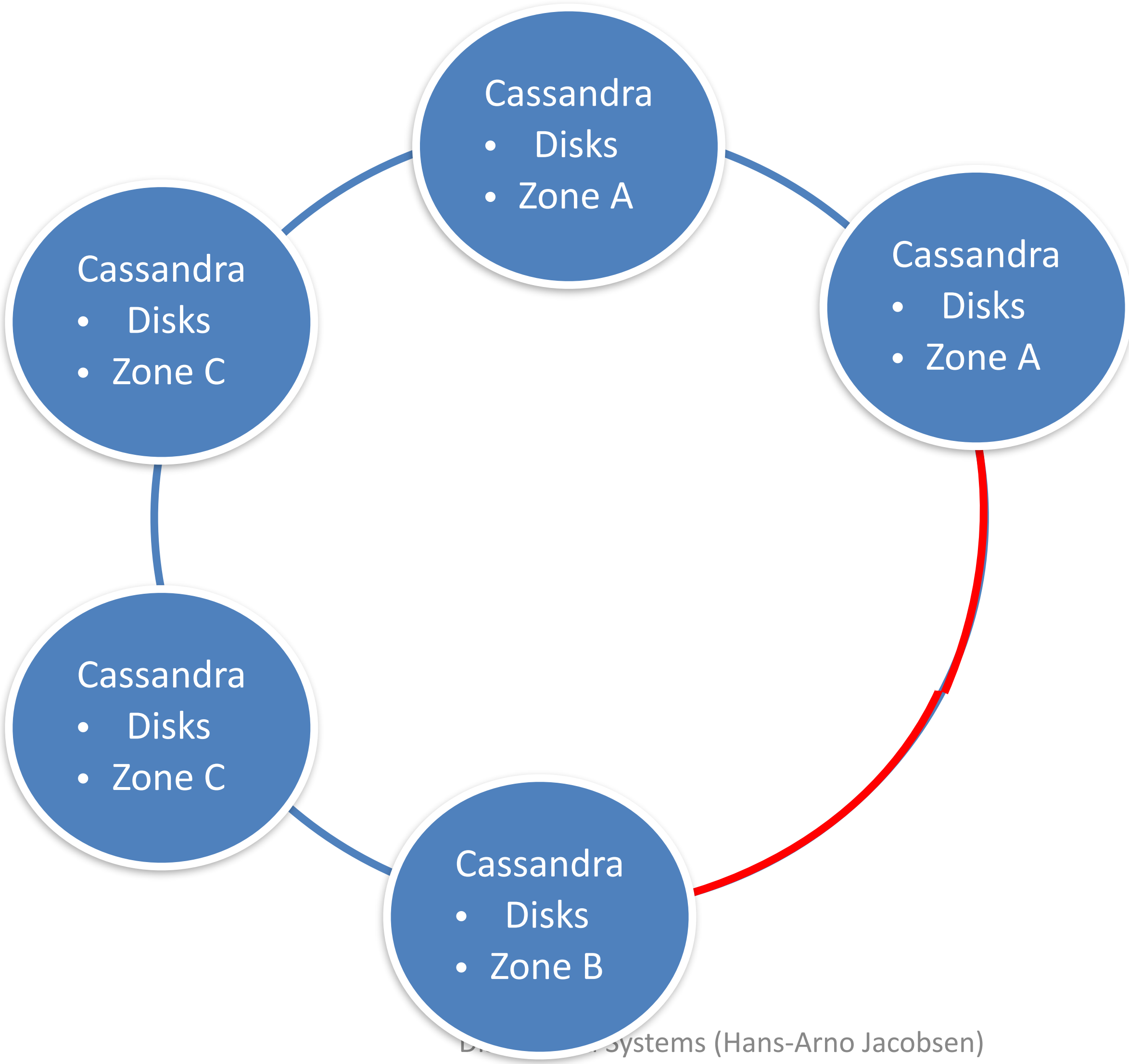
## (i.e., adding a storage unit)



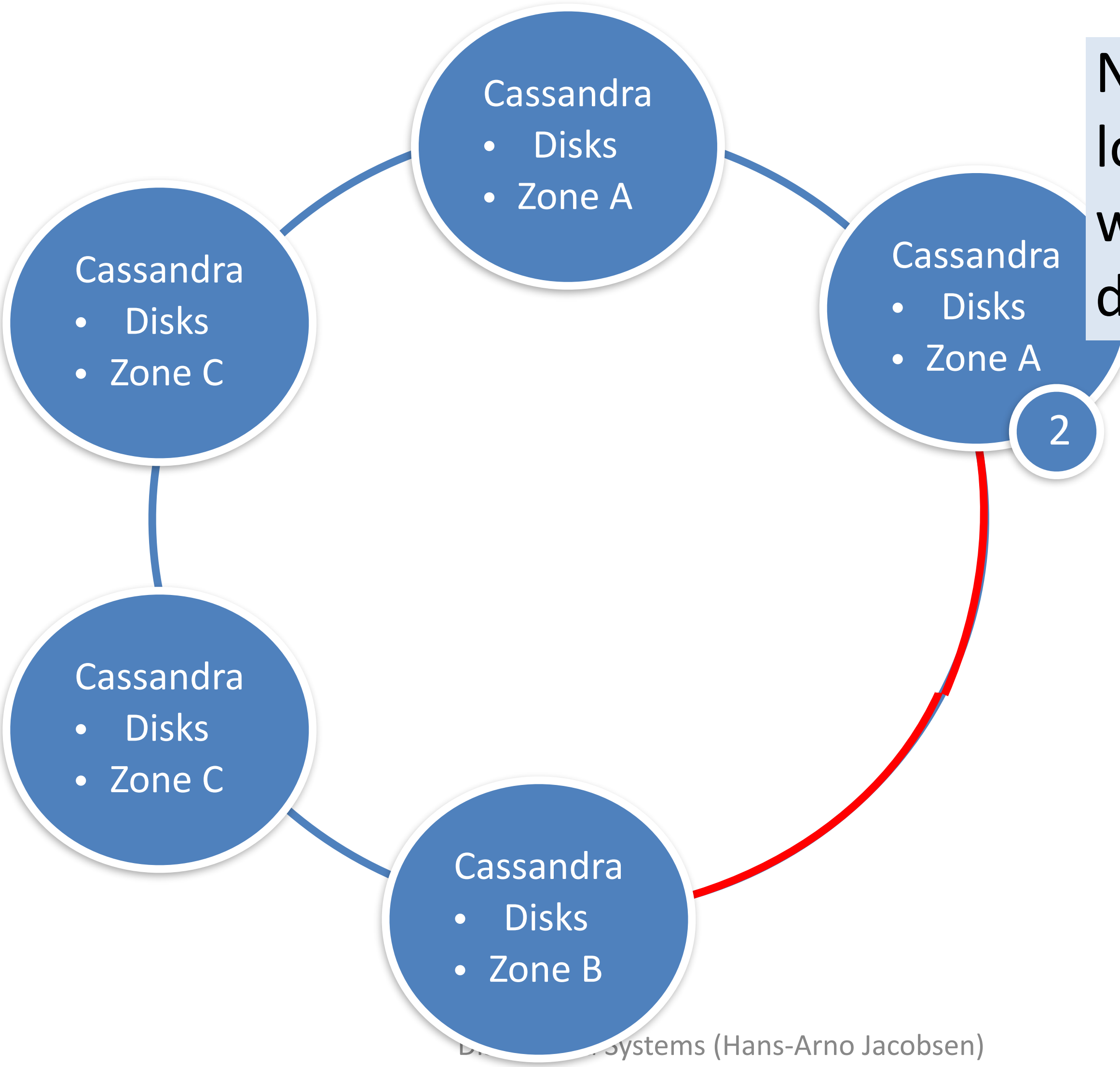
# Storage unit failure



# Storage unit failure

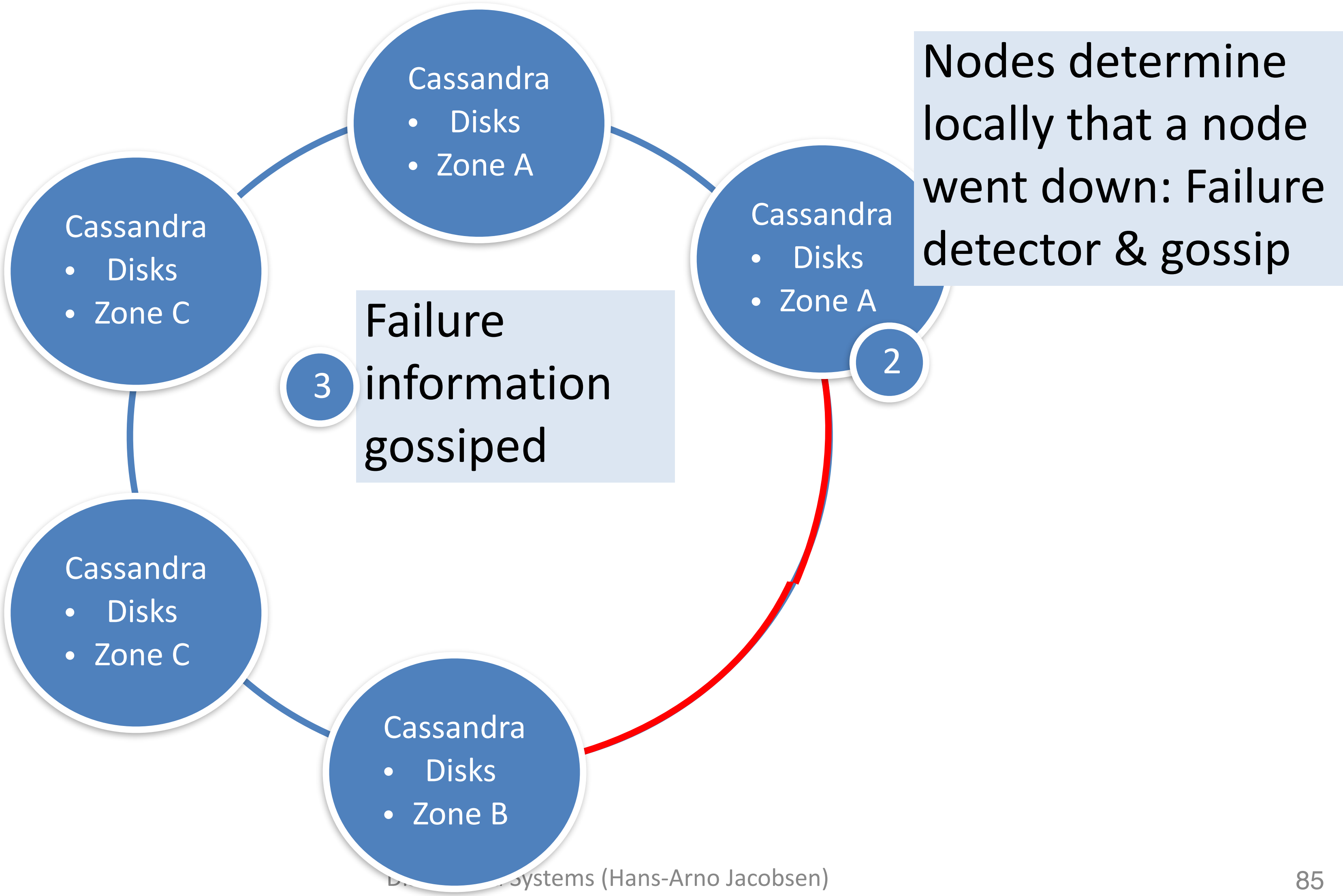


# Storage unit failure




Nodes determine locally that a node went down: Failure detector & gossip

# Storage unit failure



# Core mechanisms

- Decentralized load balancing and scalability
  - *Cf. Consistent Hashing*
- Read/write reliability
  - *Cf. Replication*
- Membership management
  - *Cf. Gossip in Replication*
- Eventual consistency model
  - *Cf. Consistency*

# Recommended Reading Materials


1. D. Karger, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663 , 1997.

2. Cassandra by example (slides):

https://de.slideshare.net/grro/cassandra-by-example-the-path-of-read-and-write-requests