COEN6731 Distributed Software Systems

Week 7: Consistent Hashing and CAP

Gengrui (Edward) Zhang, PhD
Web: gengruizhang.com

Today's outline

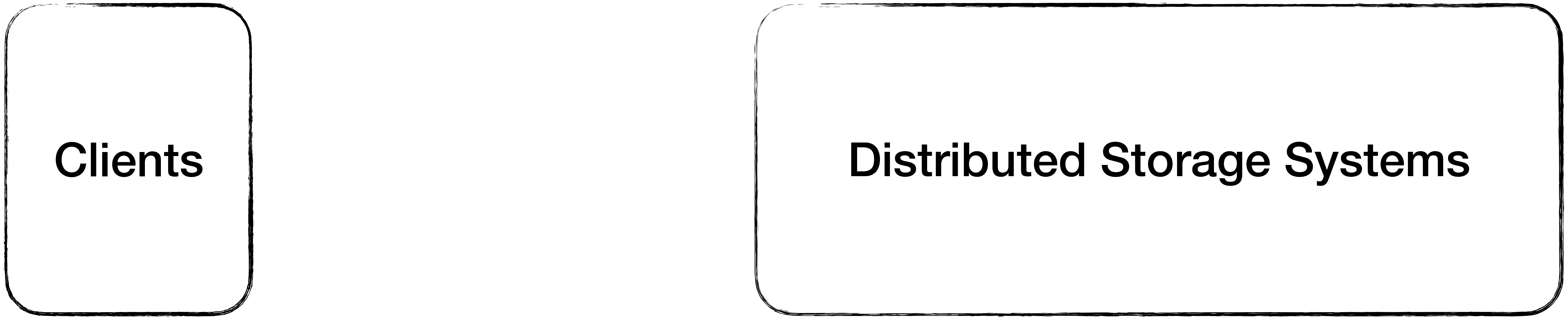
Consistent Hashing

CAP

Let’s build a distributed storage system

We aim to build a **distributed key–value storage** service that:

- Stores a large number of key–value pairs (data) across multiple servers
- Servers collectively provide a scalable distributed storage service
- Supports at least two basic operations: PUT (store) and GET (retrieve)

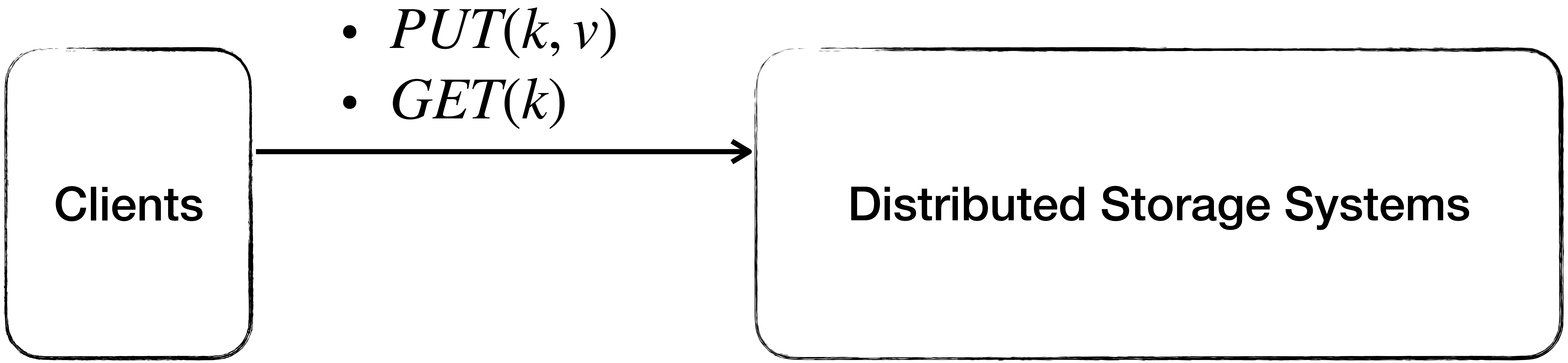


© 2025 Gengrui (Edward) Zhang

Let’s build a distributed storage system

We aim to build a **distributed key–value storage** service that:

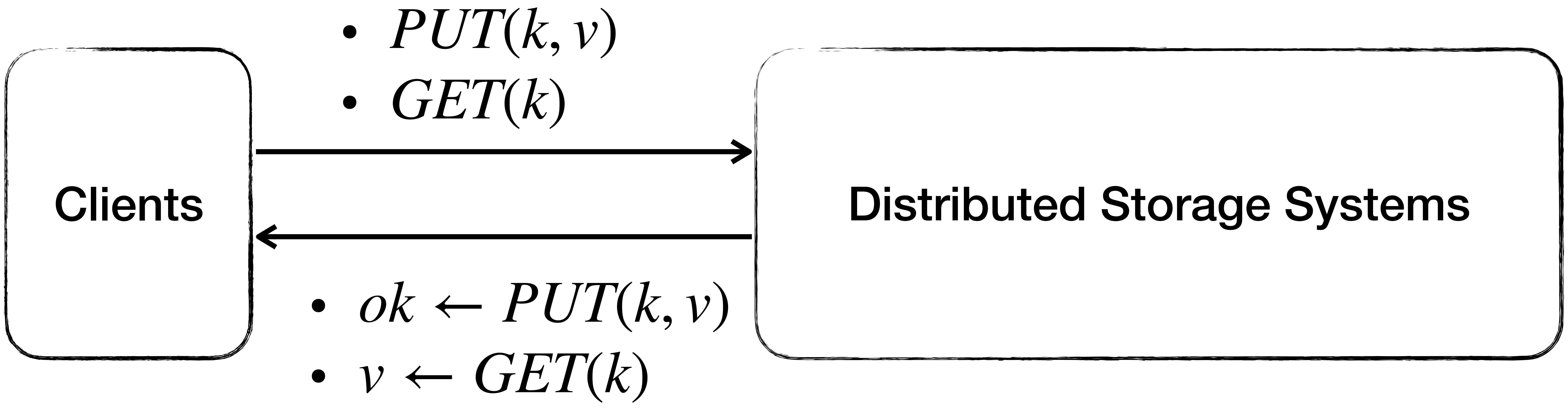
- Stores a large number of key–value pairs (data) across multiple servers
- Servers collectively provide a scalable distributed storage service
- Supports at least two basic operations: PUT (store) and GET (retrieve)



Let’s build a distributed storage system

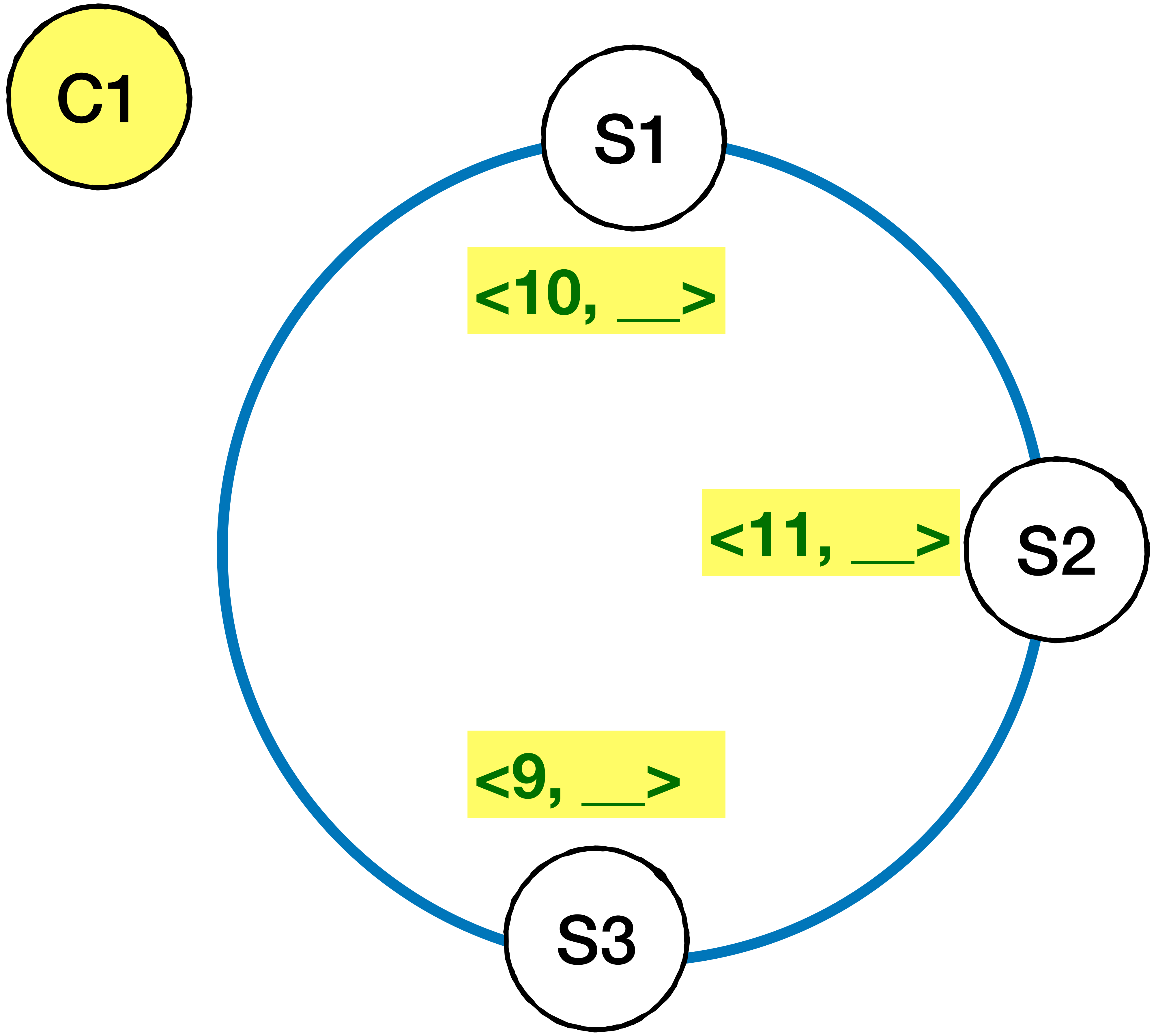
We aim to build a **distributed key–value storage** service that:

- Stores a large number of key–value pairs (data) across multiple servers
- Servers collectively provide a scalable distributed storage service
- Supports at least two basic operations: PUT (store) and GET (retrieve)



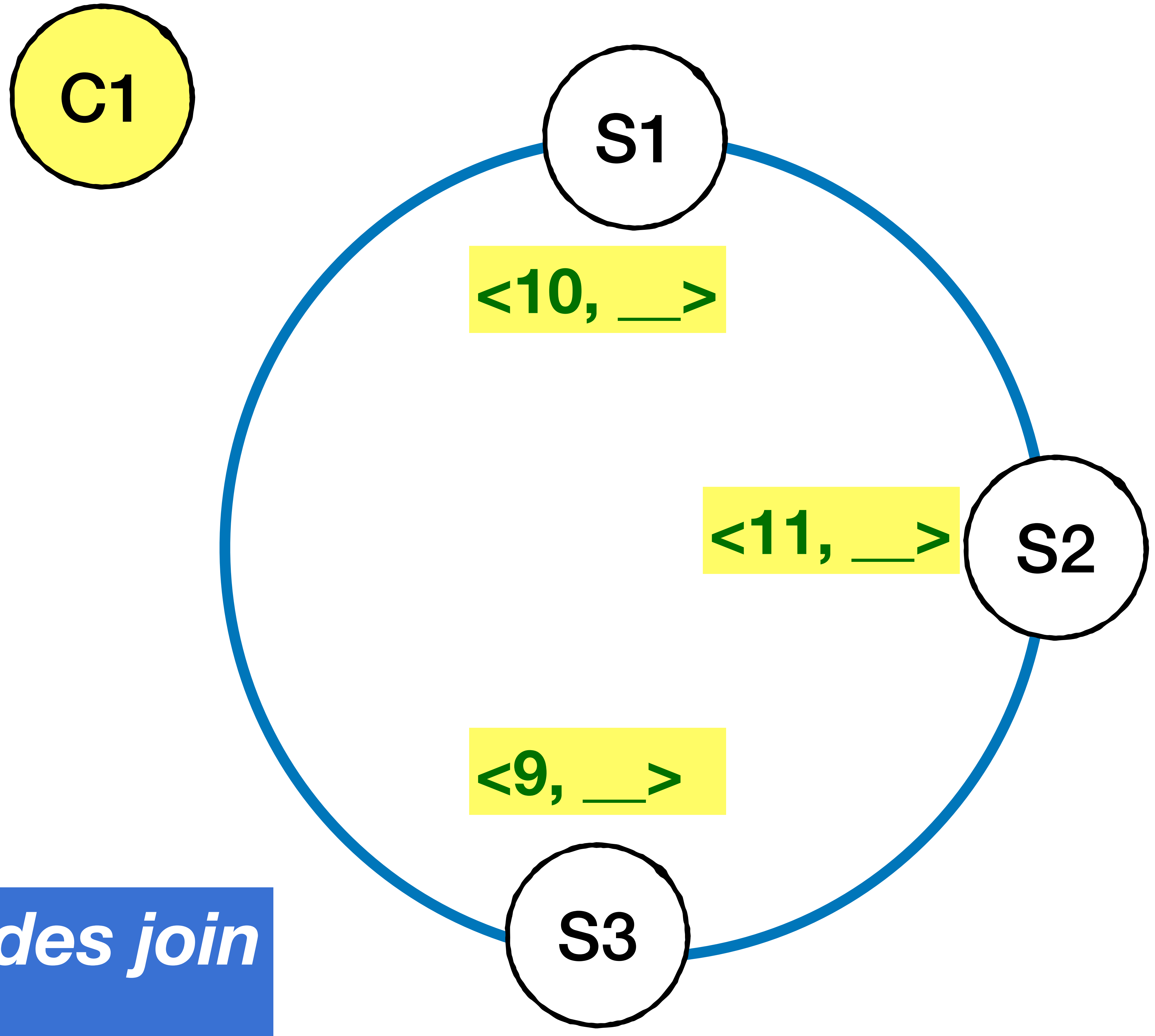
Naive Hashing

- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



Naive Hashing

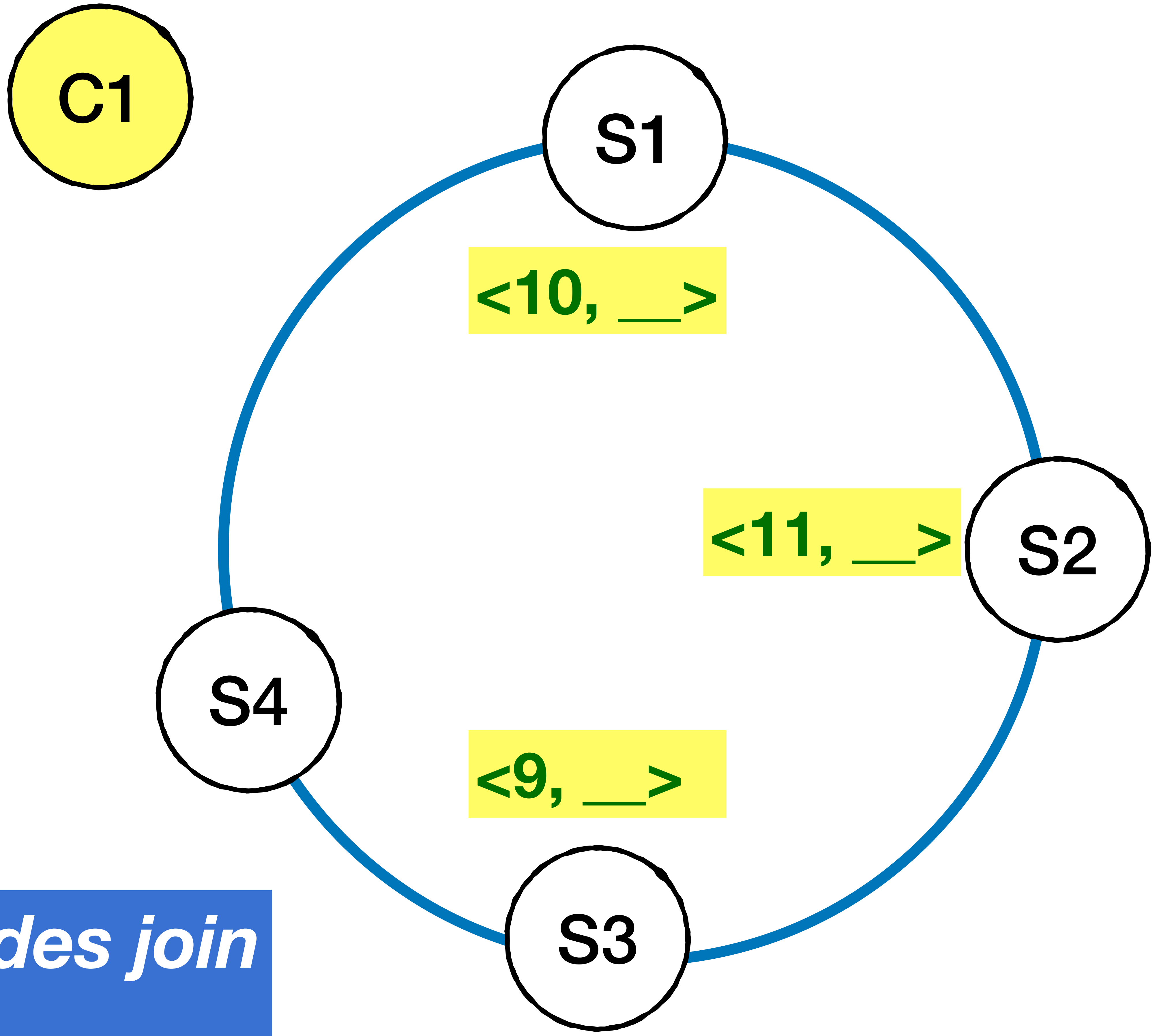
- Let's try the simple solution:
  - server = hash(key) % N
- We'd like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



What happens when nodes join or leave?

Naive Hashing

- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>

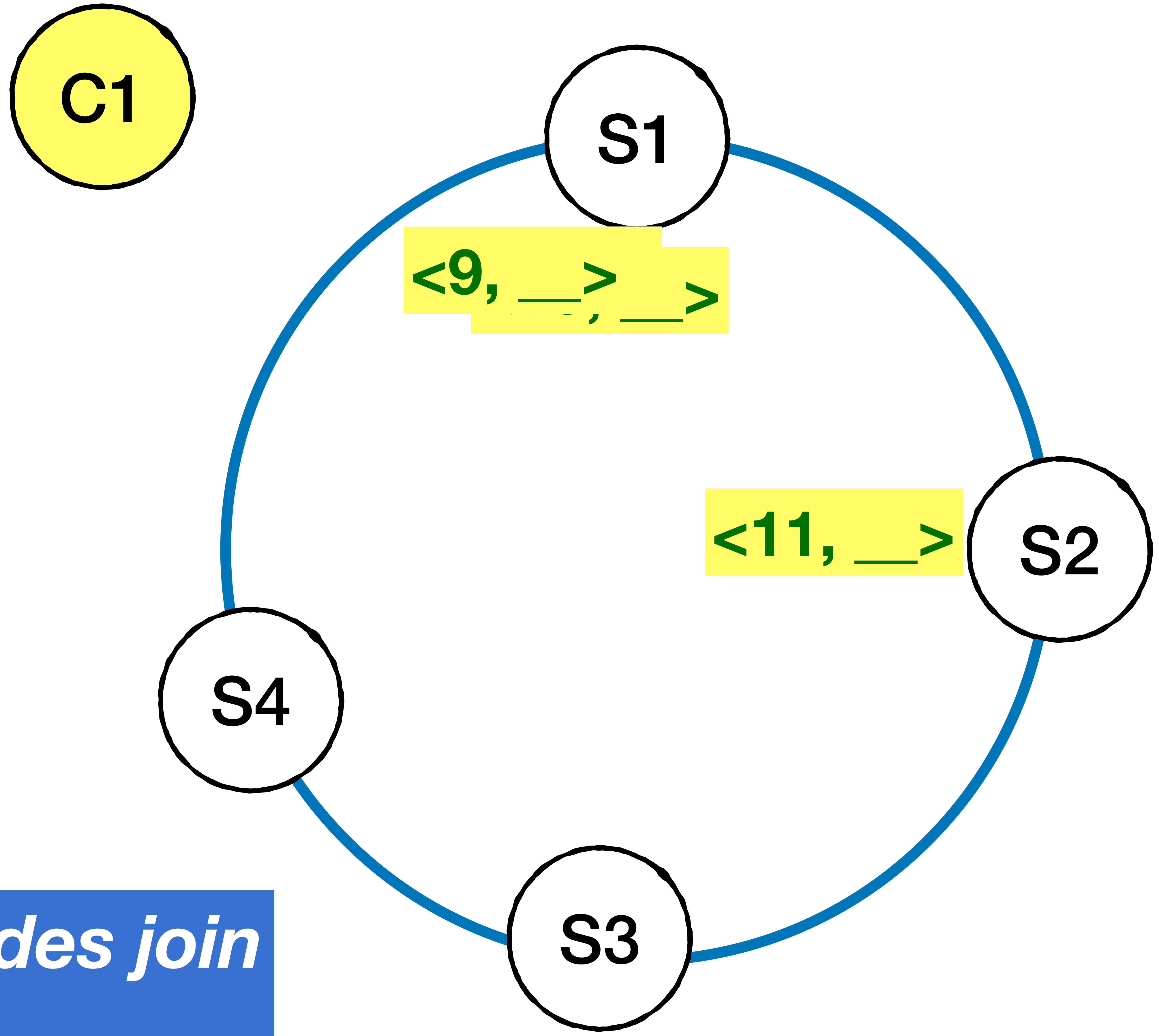


What happens when nodes join or leave?

© 2025 Gengrui (Edward) Zhang

Naive Hashing

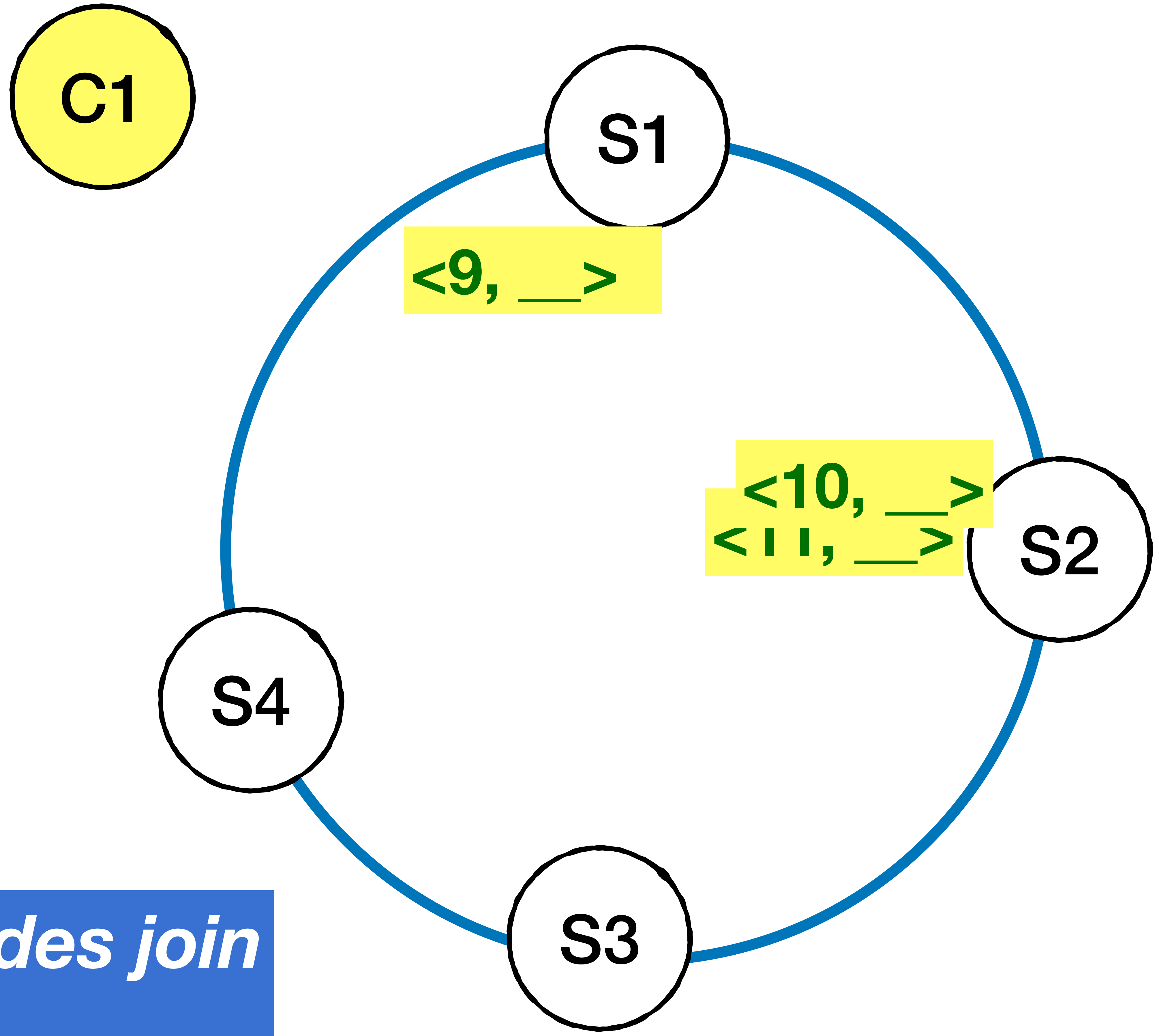
- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



What happens when nodes join or leave?

Naive Hashing

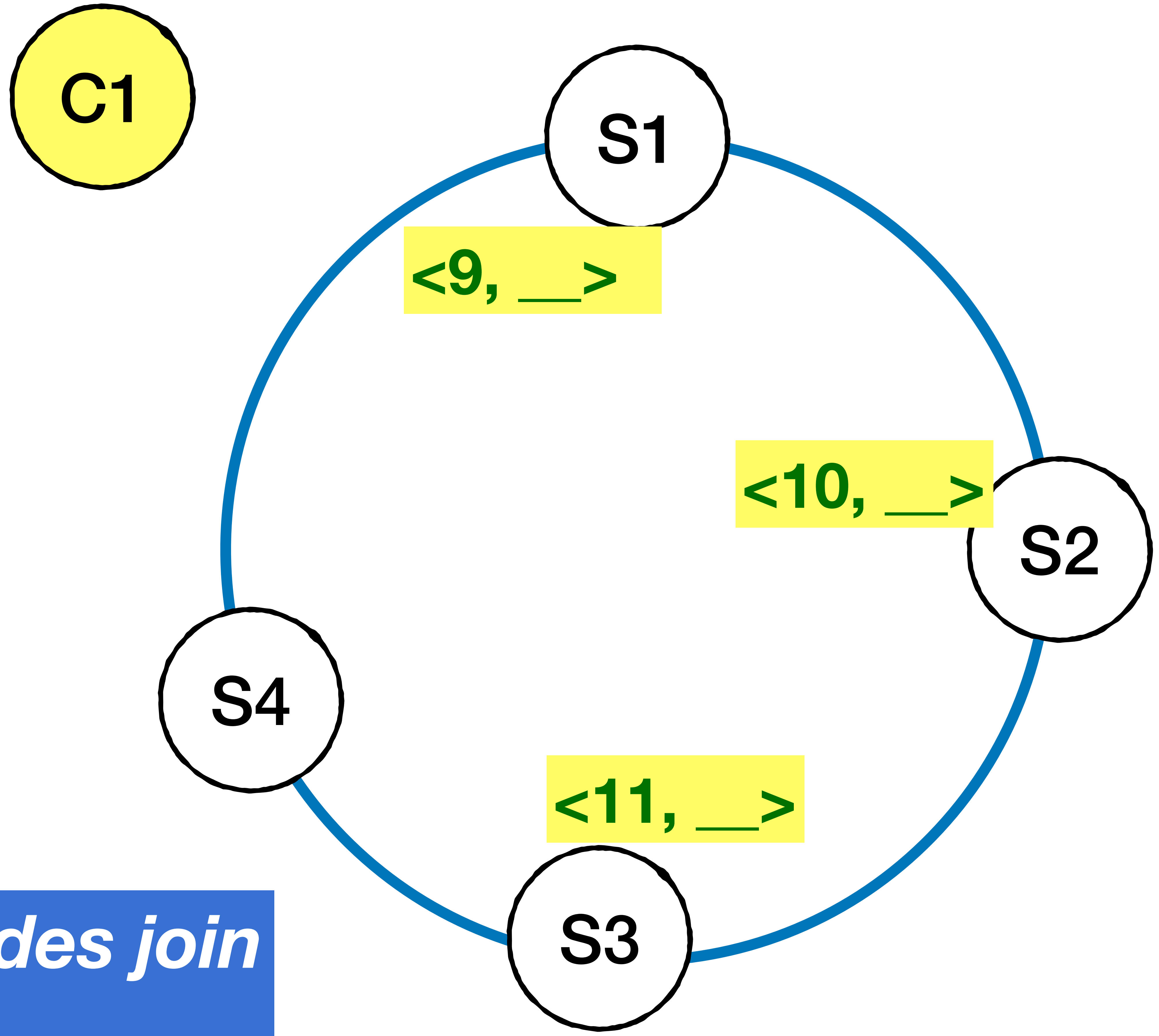
- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



What happens when nodes join or leave?

Naive Hashing

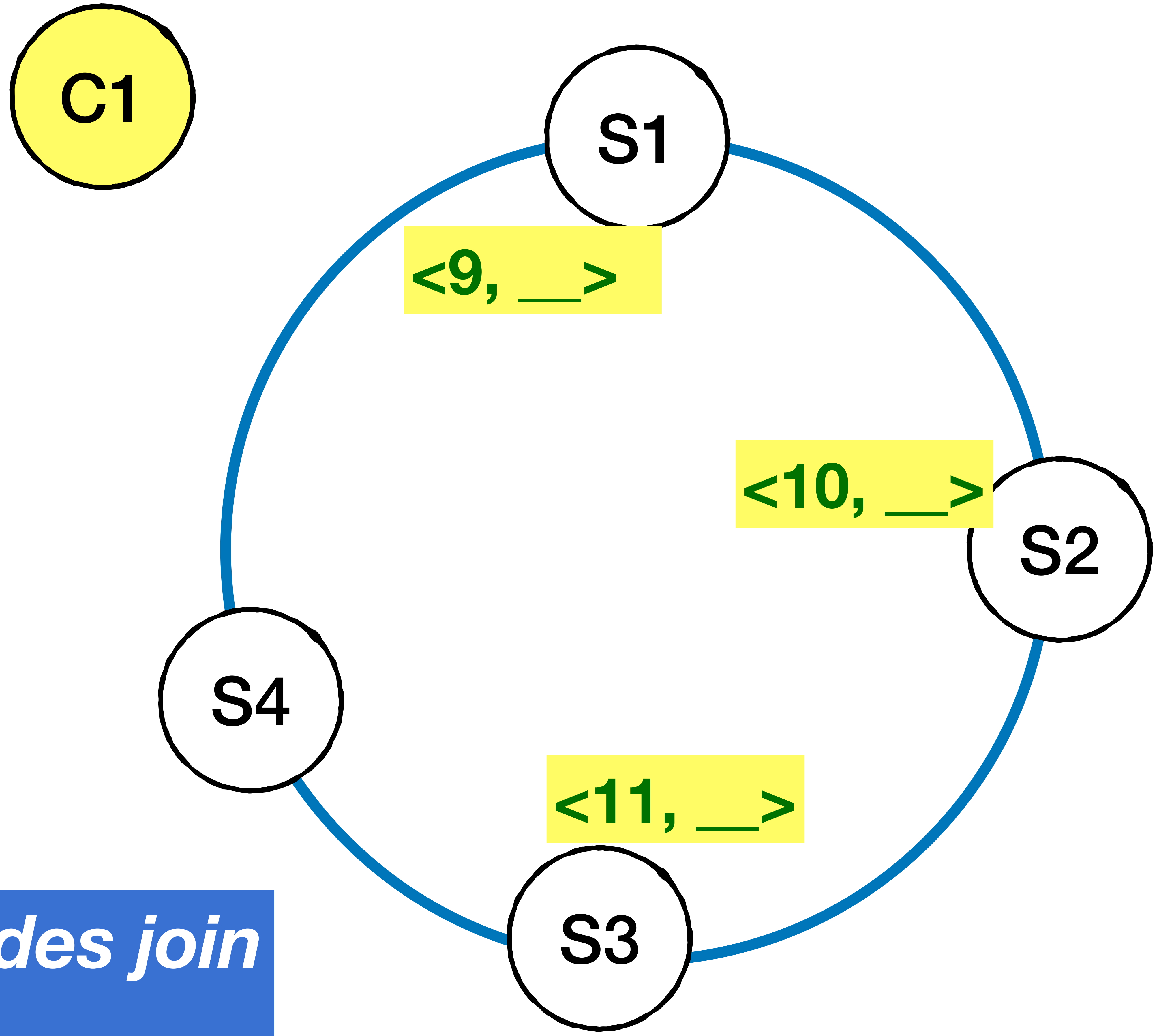
- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



What happens when nodes join or leave?

Naive Hashing

- Let’s try the simple solution:
  - server = hash(key) % N
- We’d like to operate PUT/GET
  - PUT <9, __> (key=9%3=0)
  - PUT <10, __> (key=10%3=1)
  - PUT <11, __> (key=11%3=2)
  - GET <11>
  - GET <15>



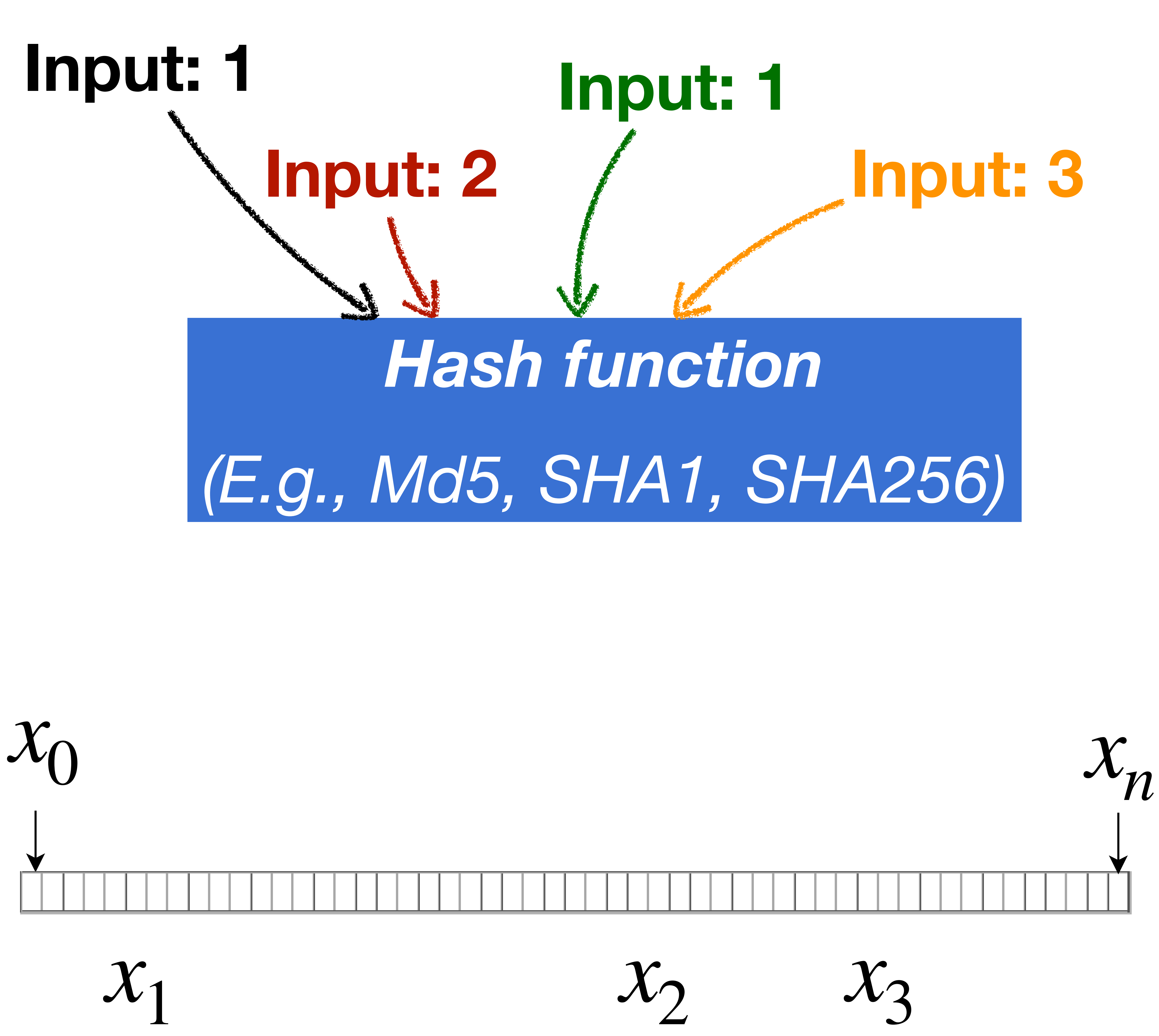
What happens when nodes join or leave?

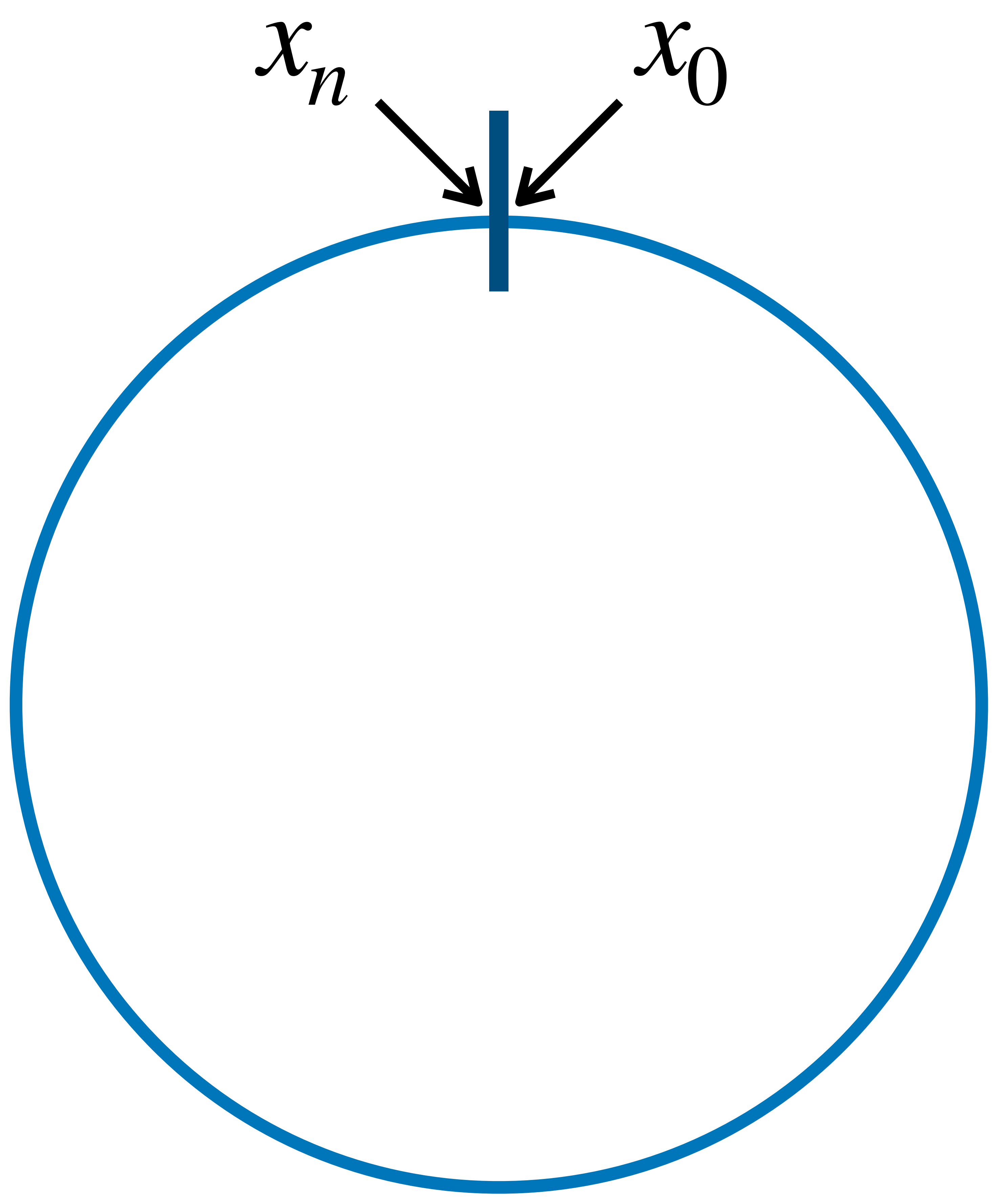
Everything moved! :(

Motivation for consistent hashing

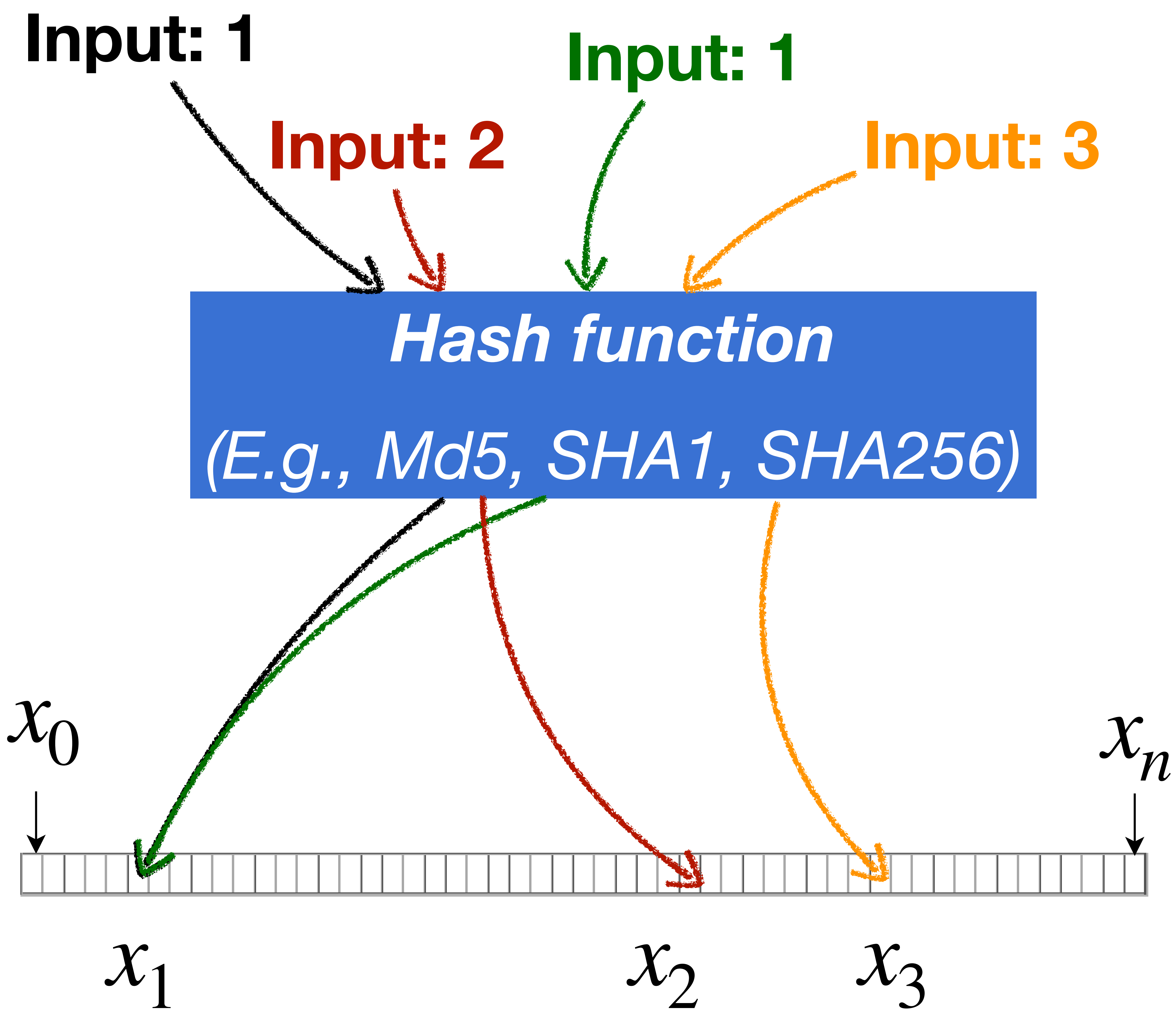
- Our goal: Distribute keys across dynamic servers while **minimizing remapping when the number of servers changes**
  - Balance load across servers
  - Minimize key movement when nodes join/leave
  - Support elasticity (scaling up/down)
  - Localize failure impact

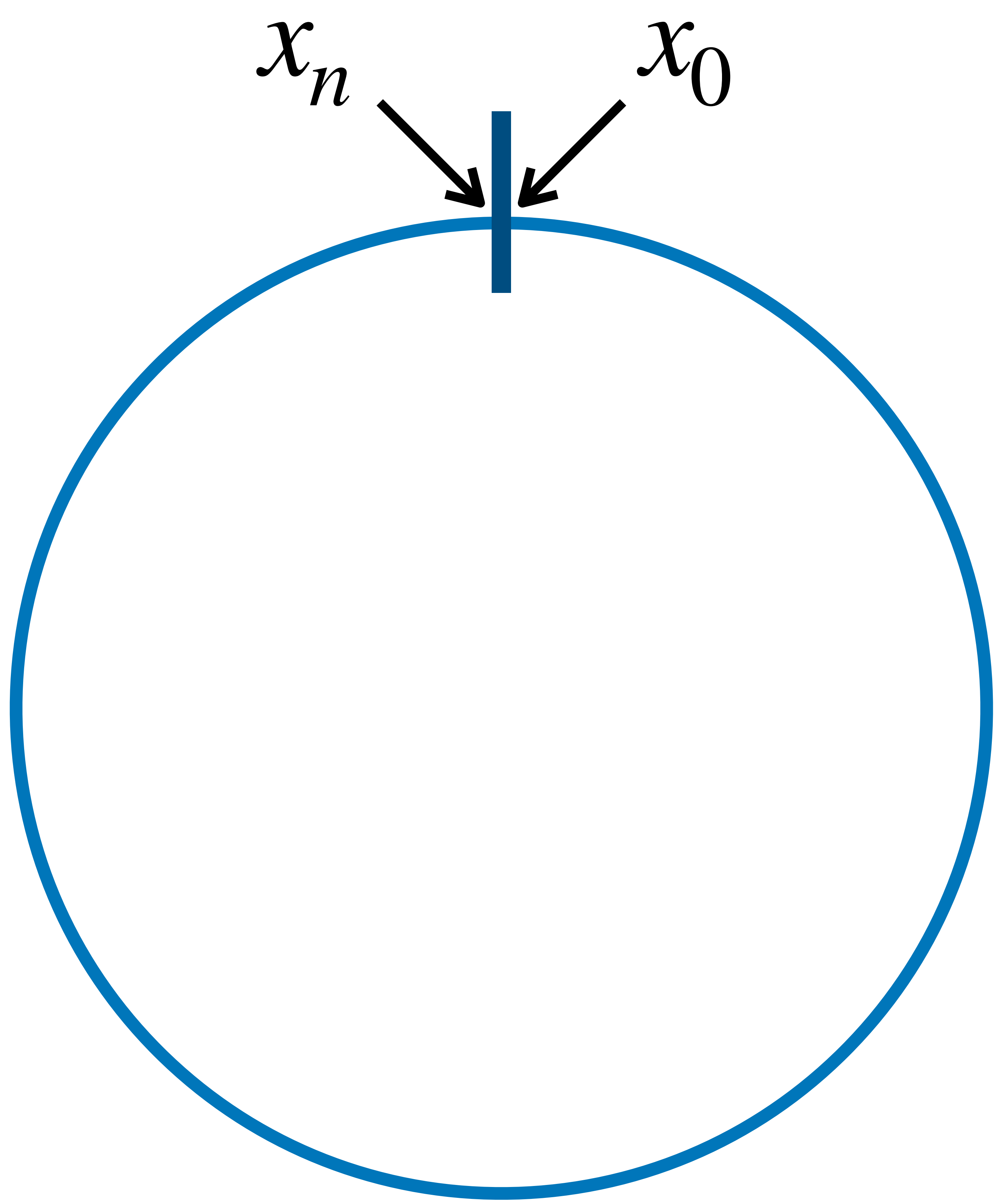
Hash space and hash ring



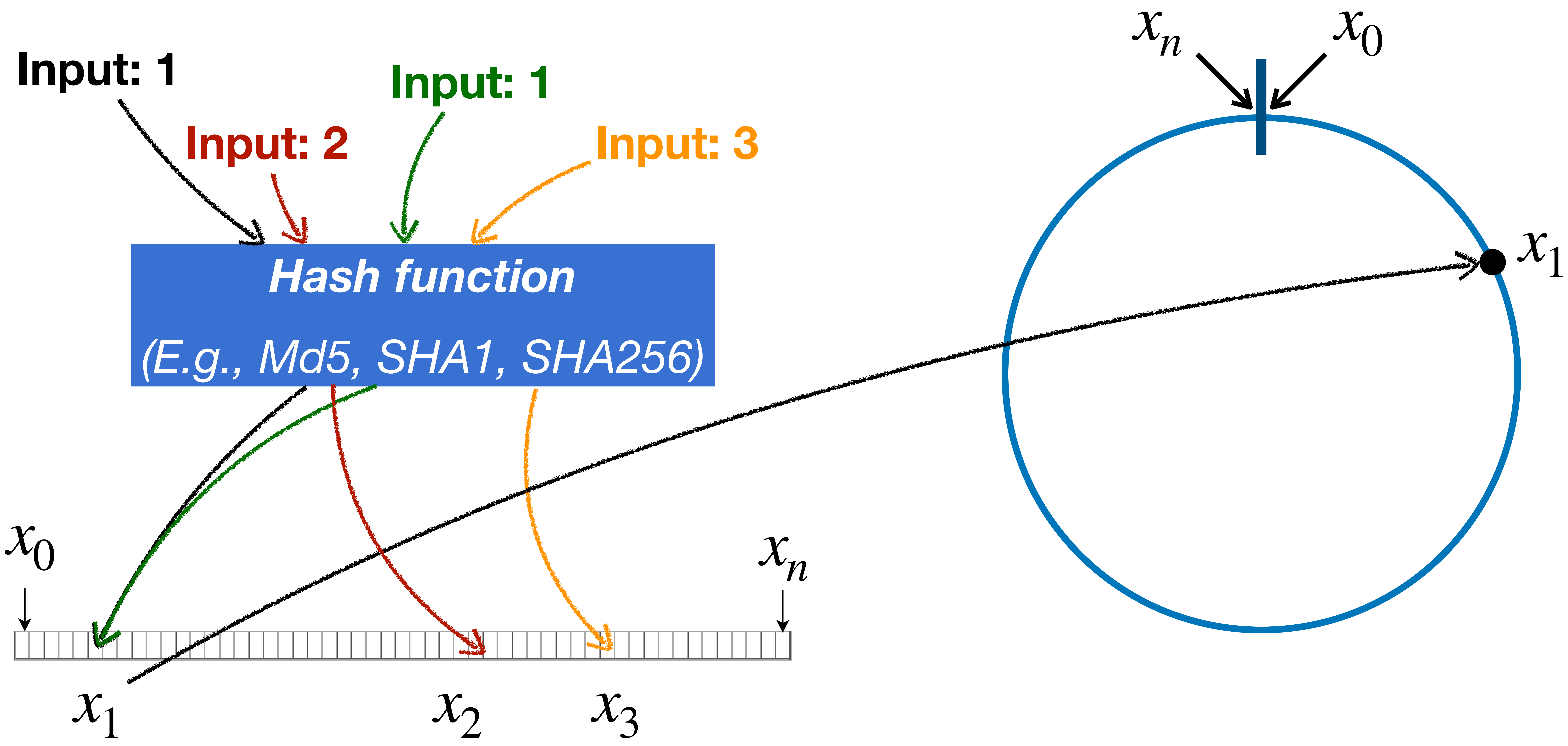


Hash space and hash ring

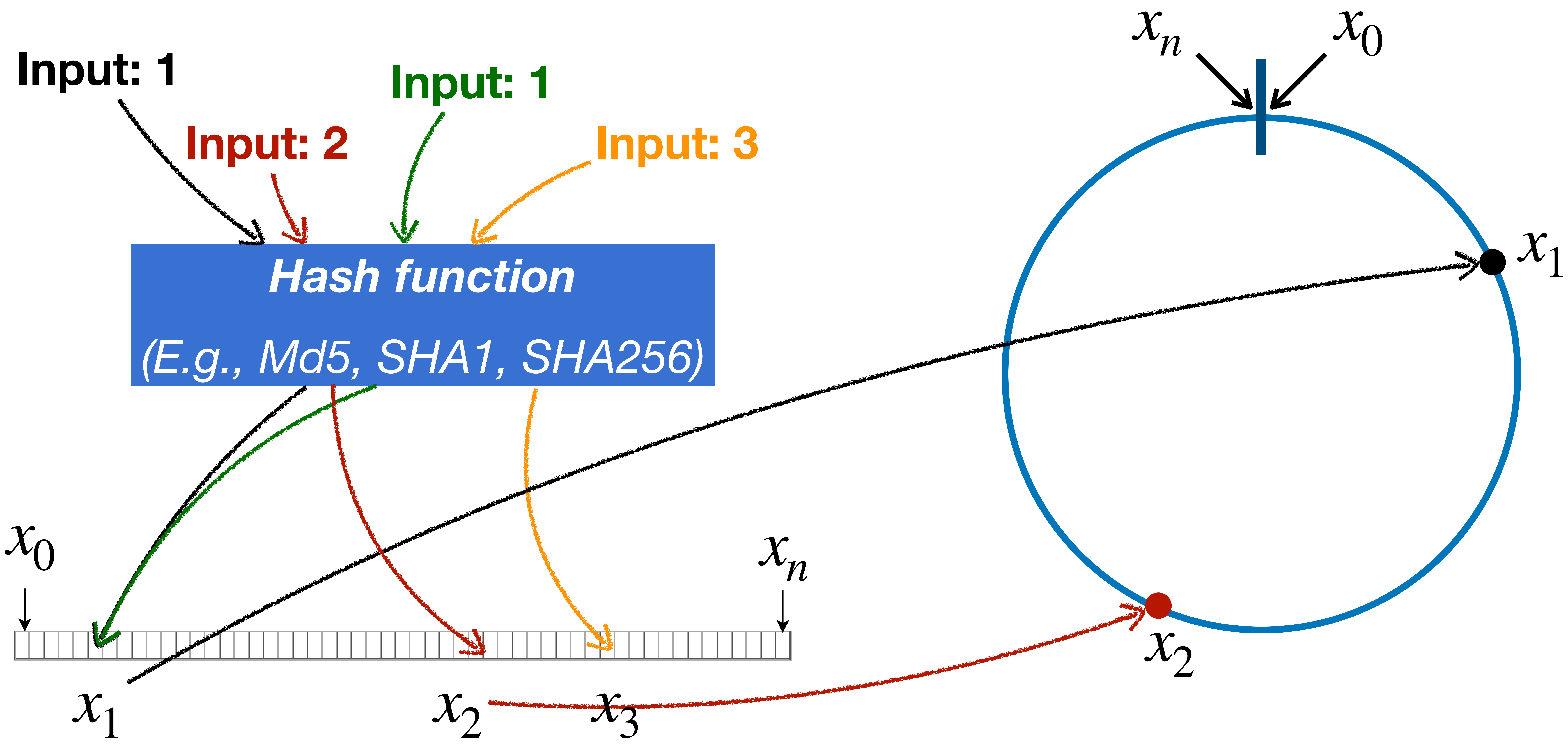




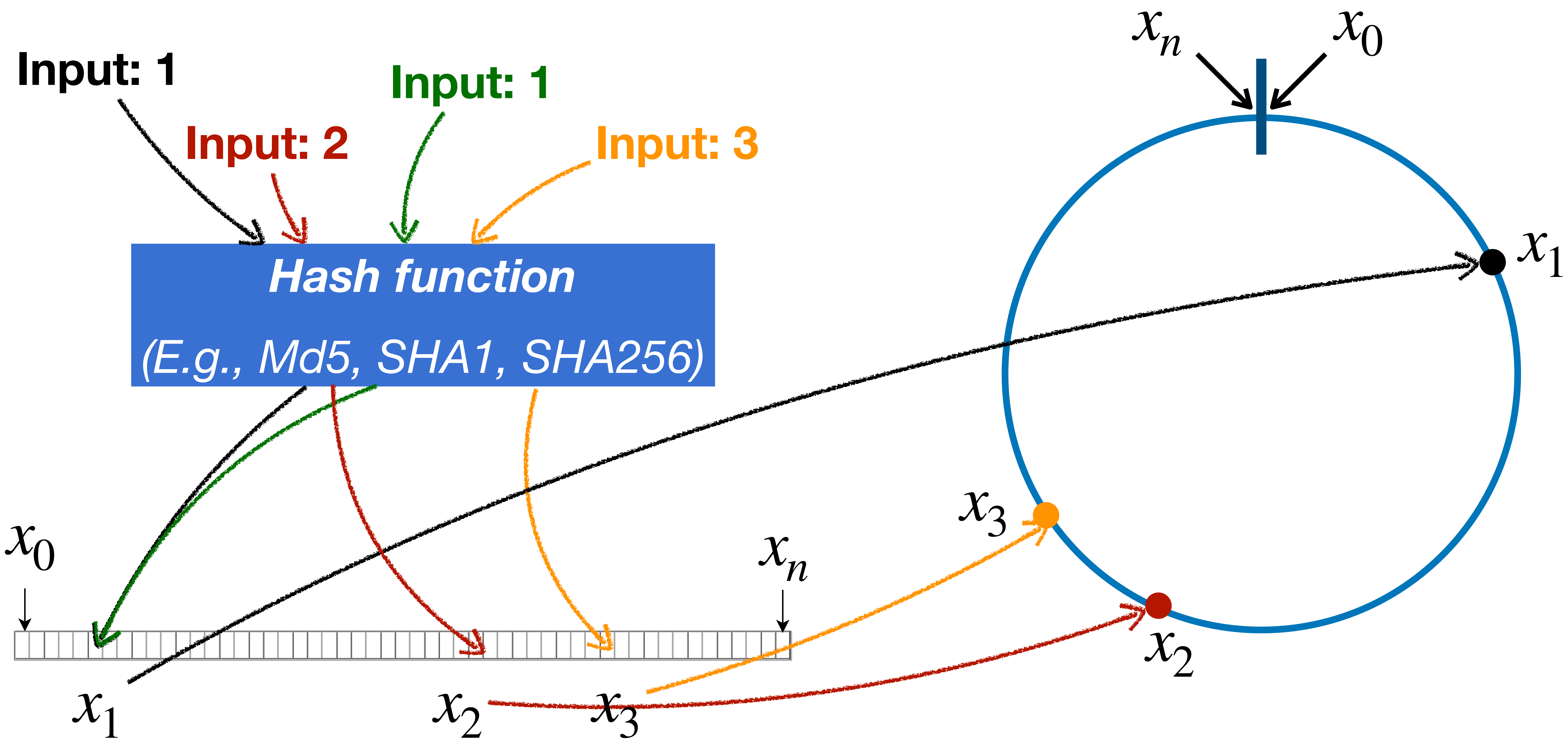
Hash space and hash ring



Hash space and hash ring



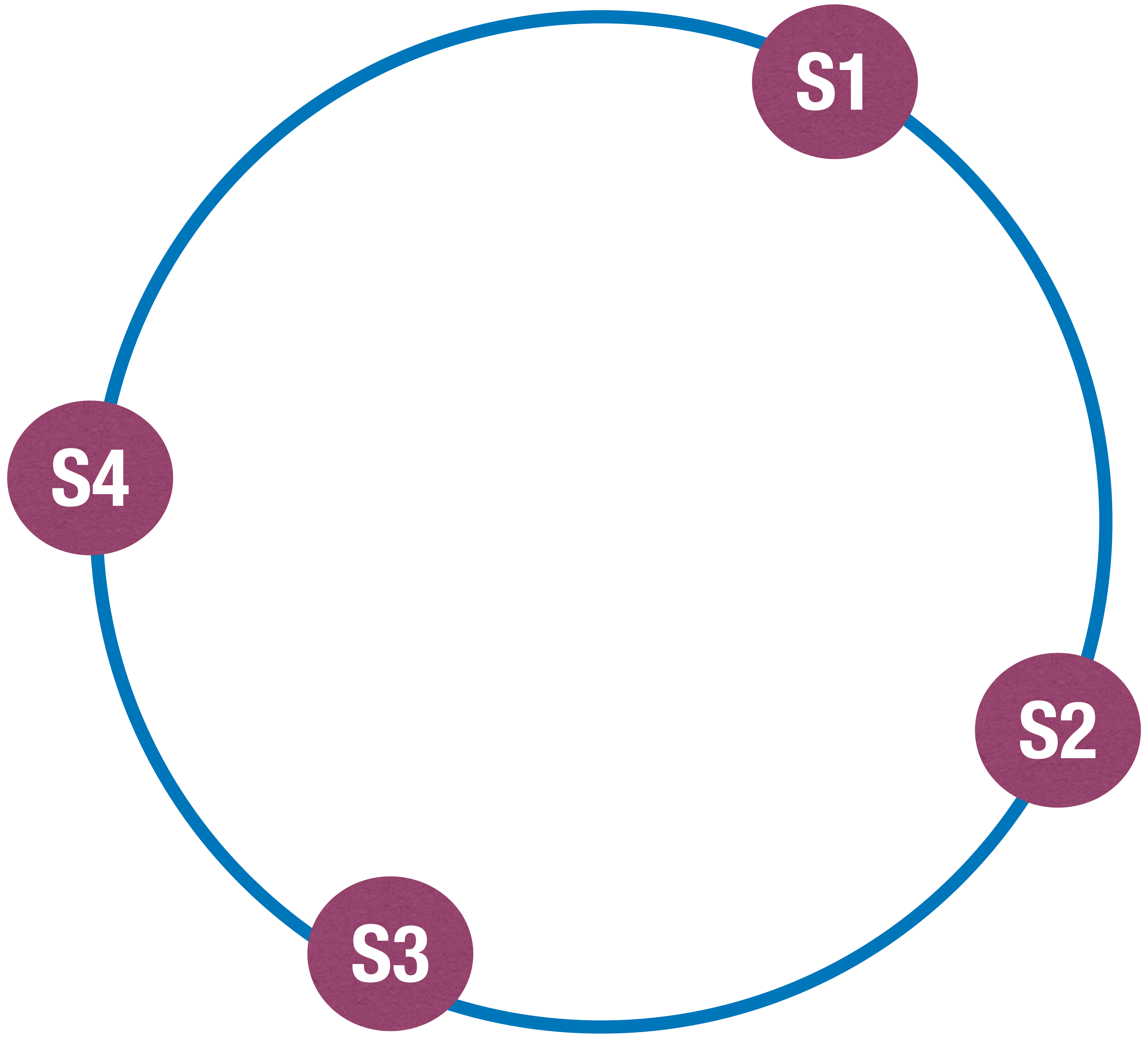
Hash space and hash ring



Consistent hashing

Step 1: Hash servers

- (E.g, using their IPs)

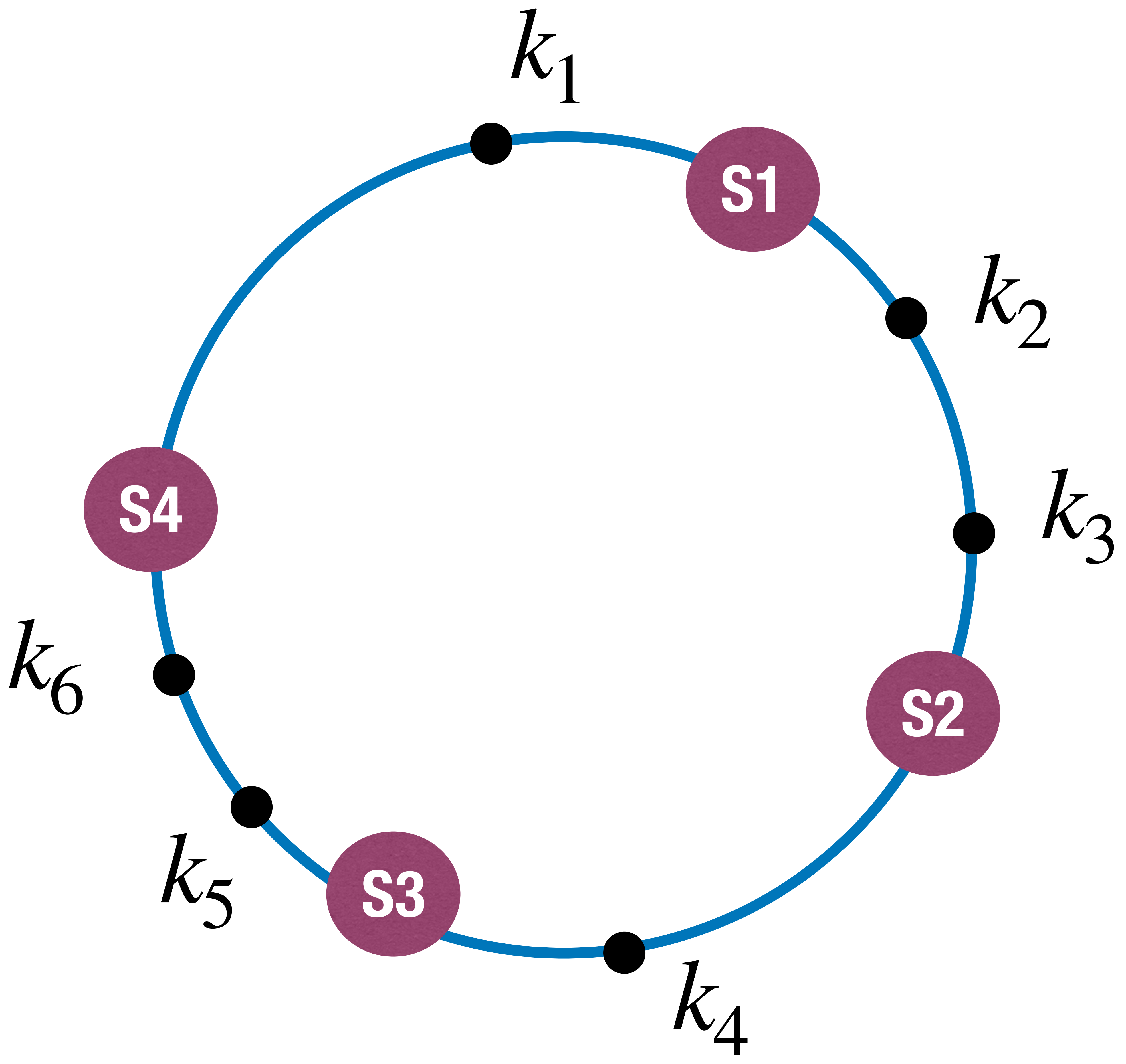


Consistent hashing

Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys



Consistent hashing

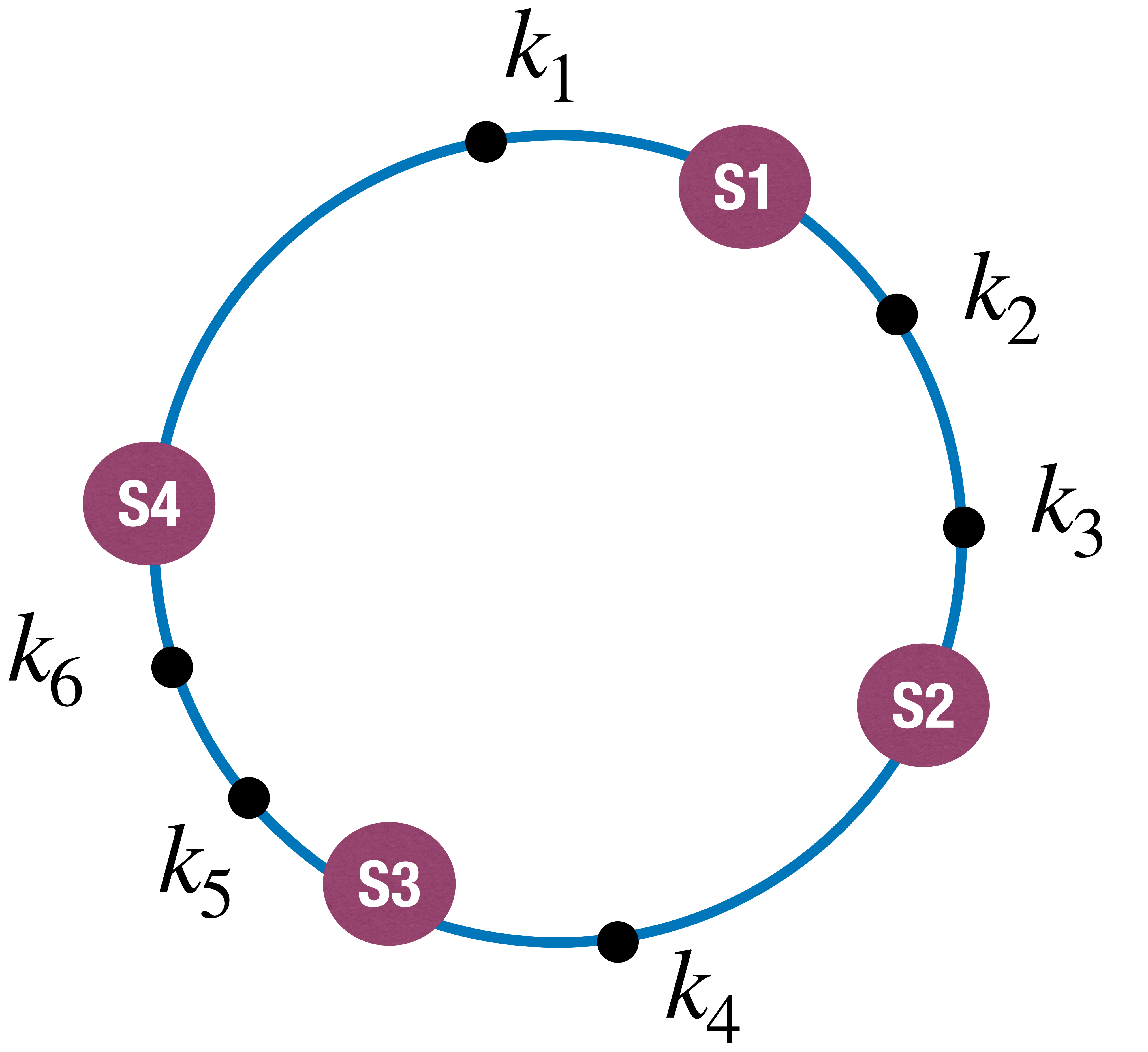
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



© 2025 Gengrui (Edward) Zhang

Consistent hashing

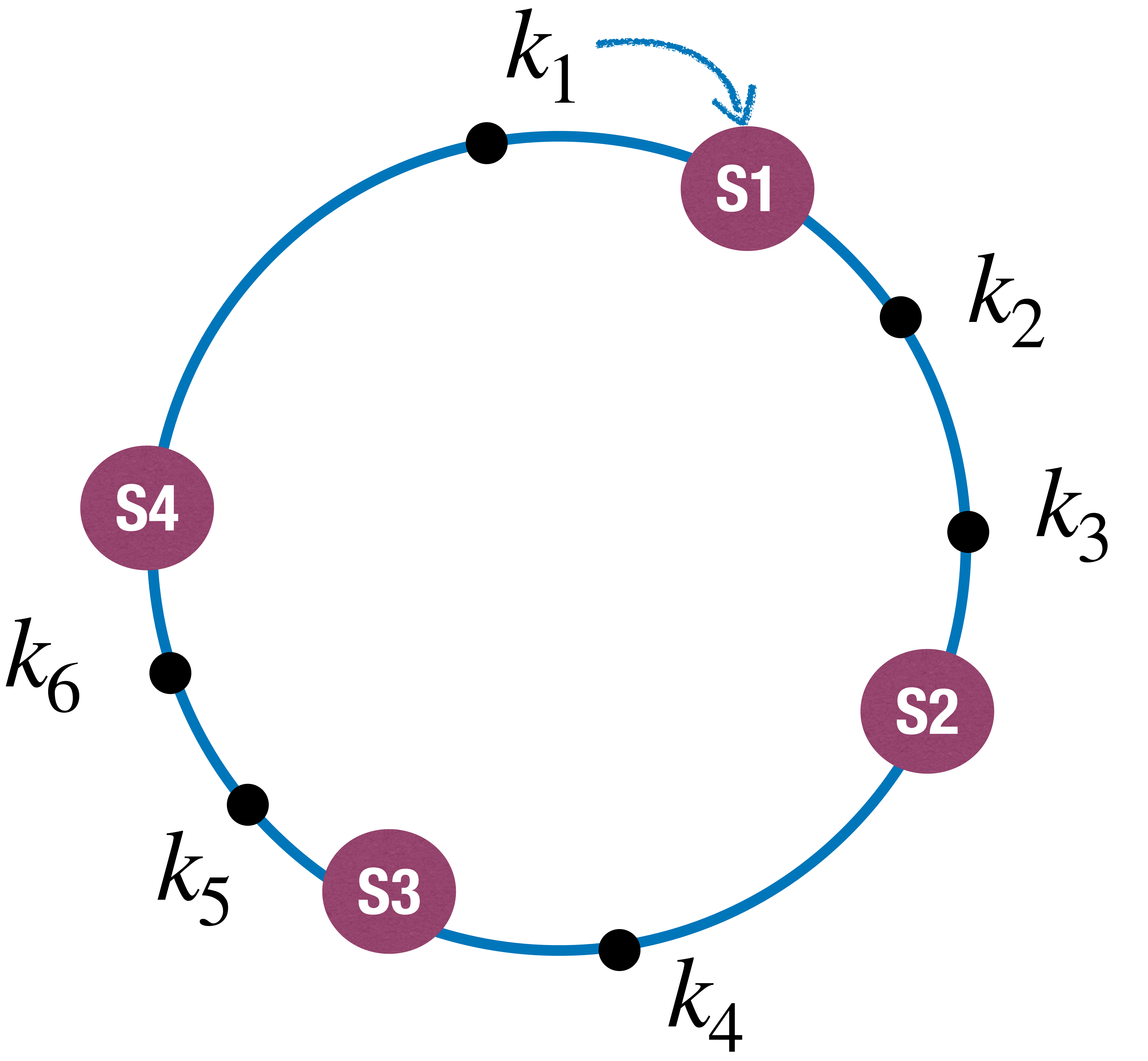
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



Consistent hashing

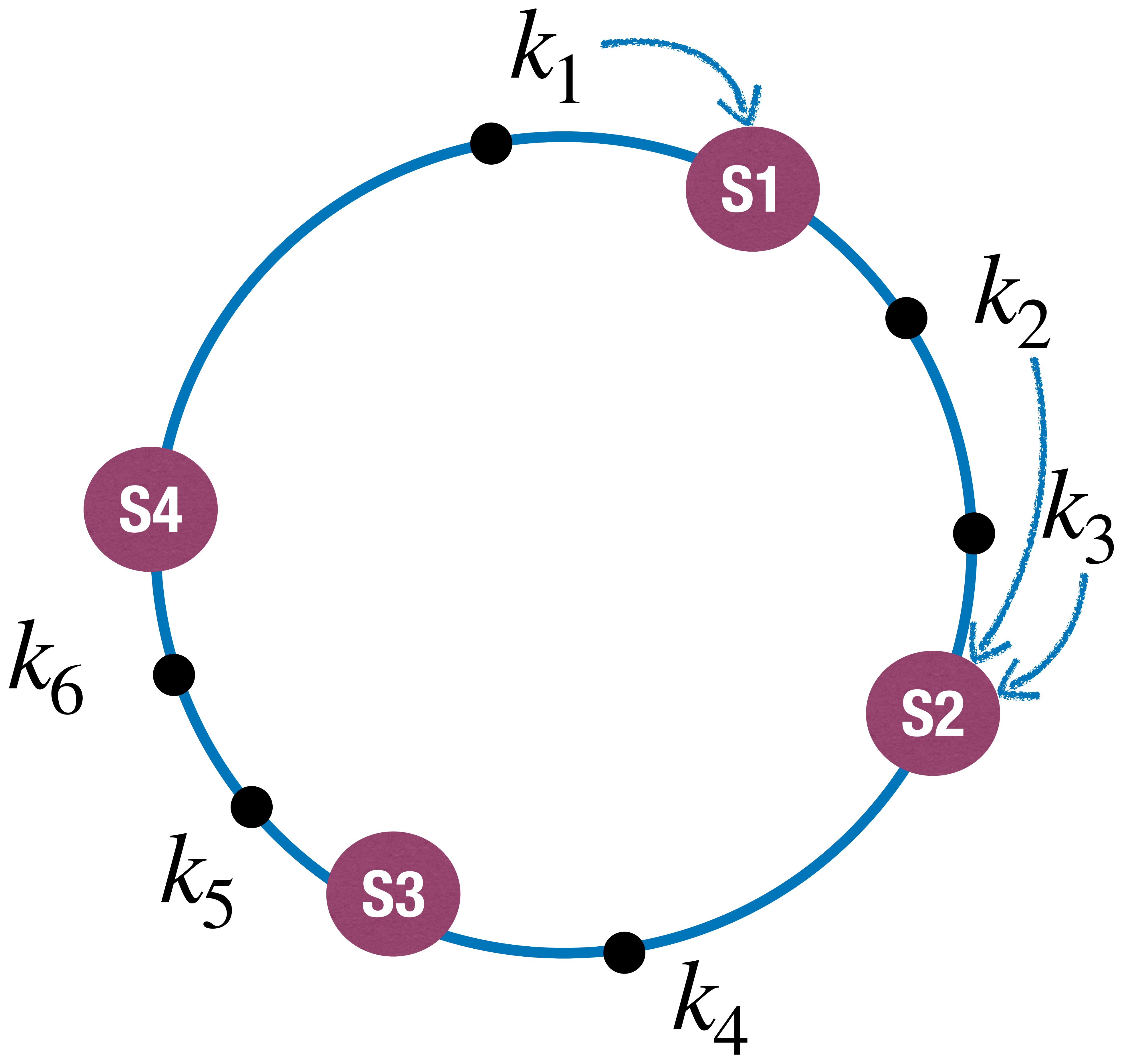
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



Consistent hashing

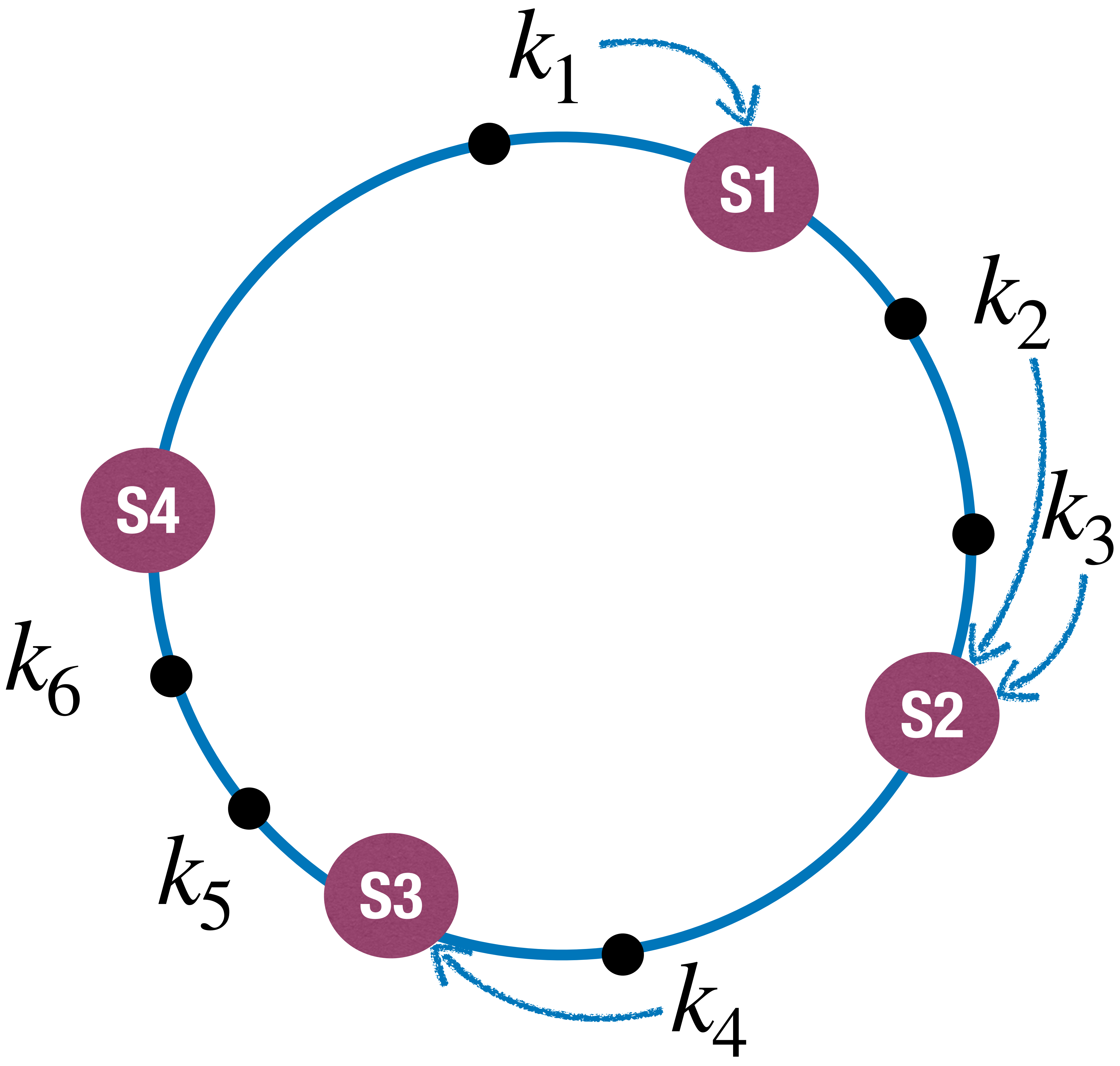
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



Consistent hashing

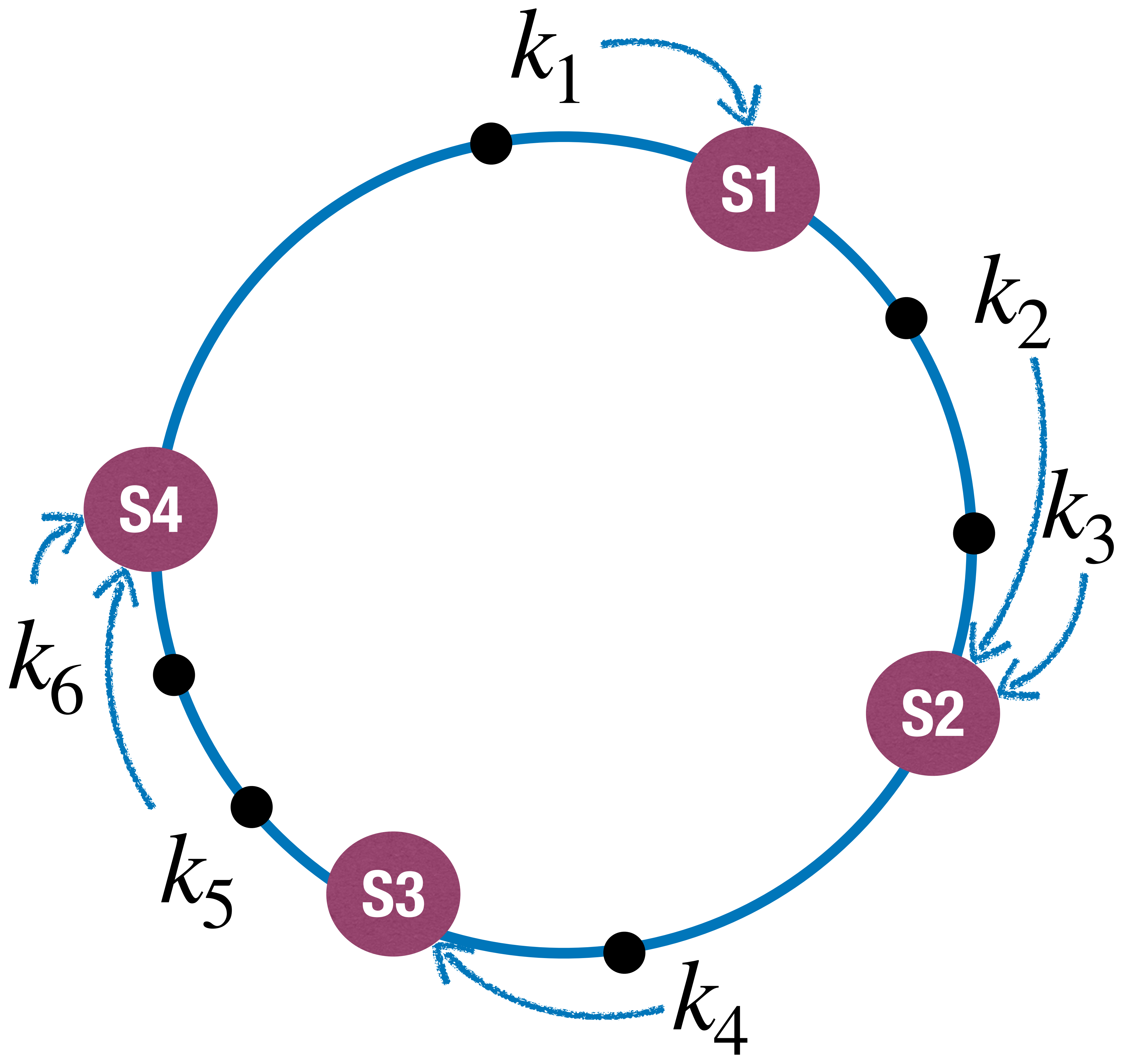
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



Consistent hashing

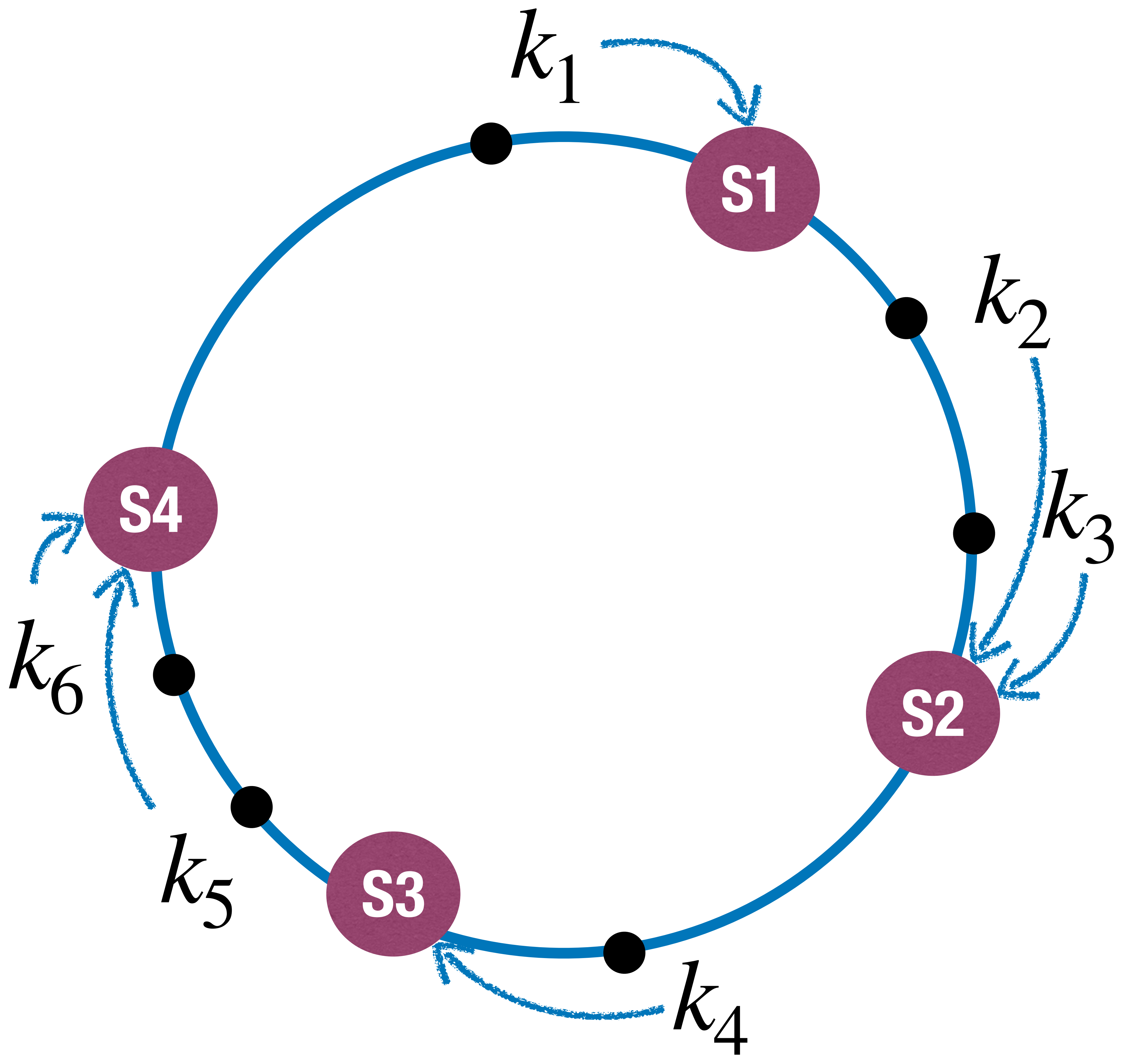
Step 1: Hash servers

- (E.g, using their IPs)

Step 2: Hash keys

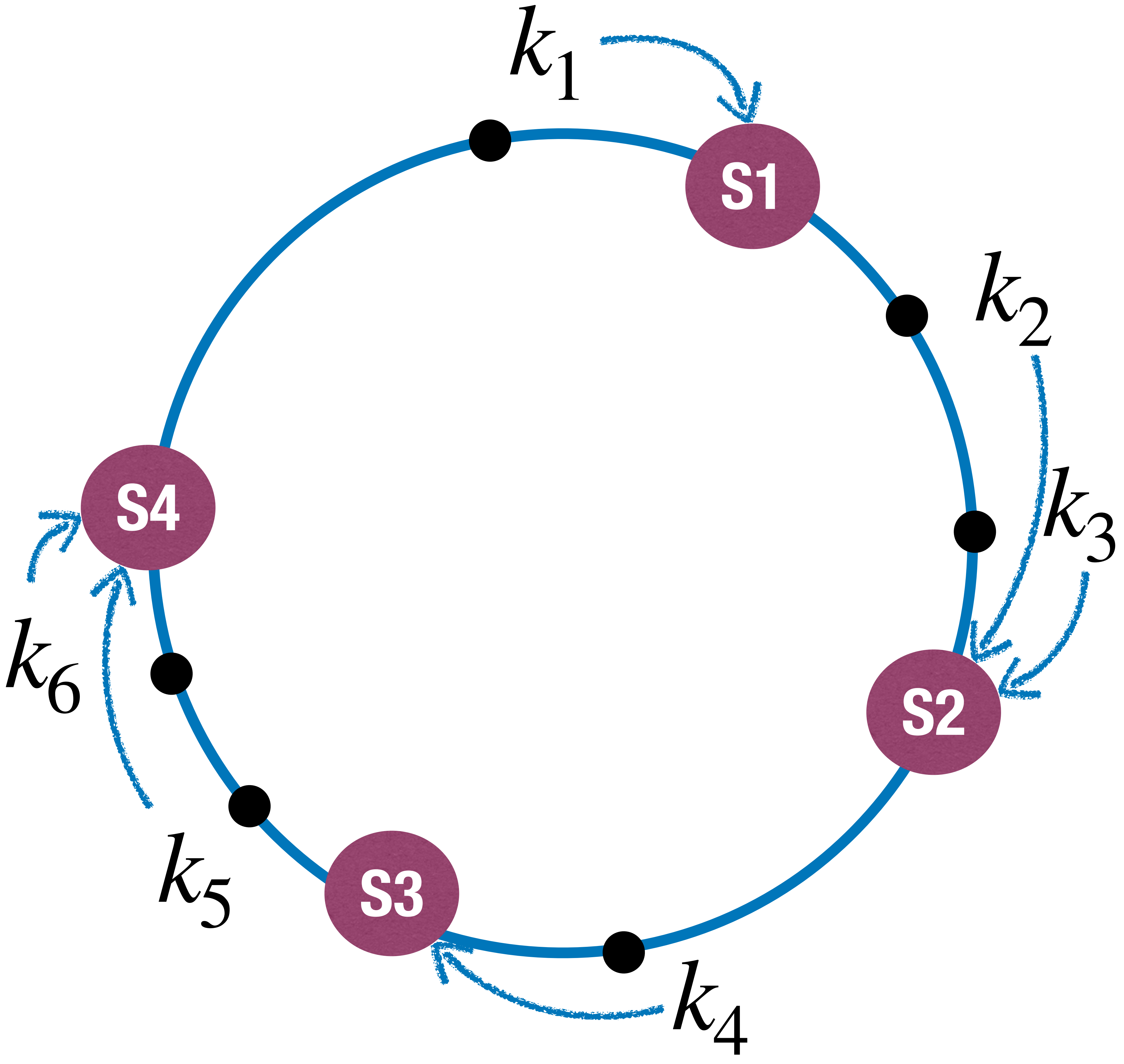
Step 3: Server look up

To determine which server a key is stored on, go clockwise from the key position on the ring until a server is found



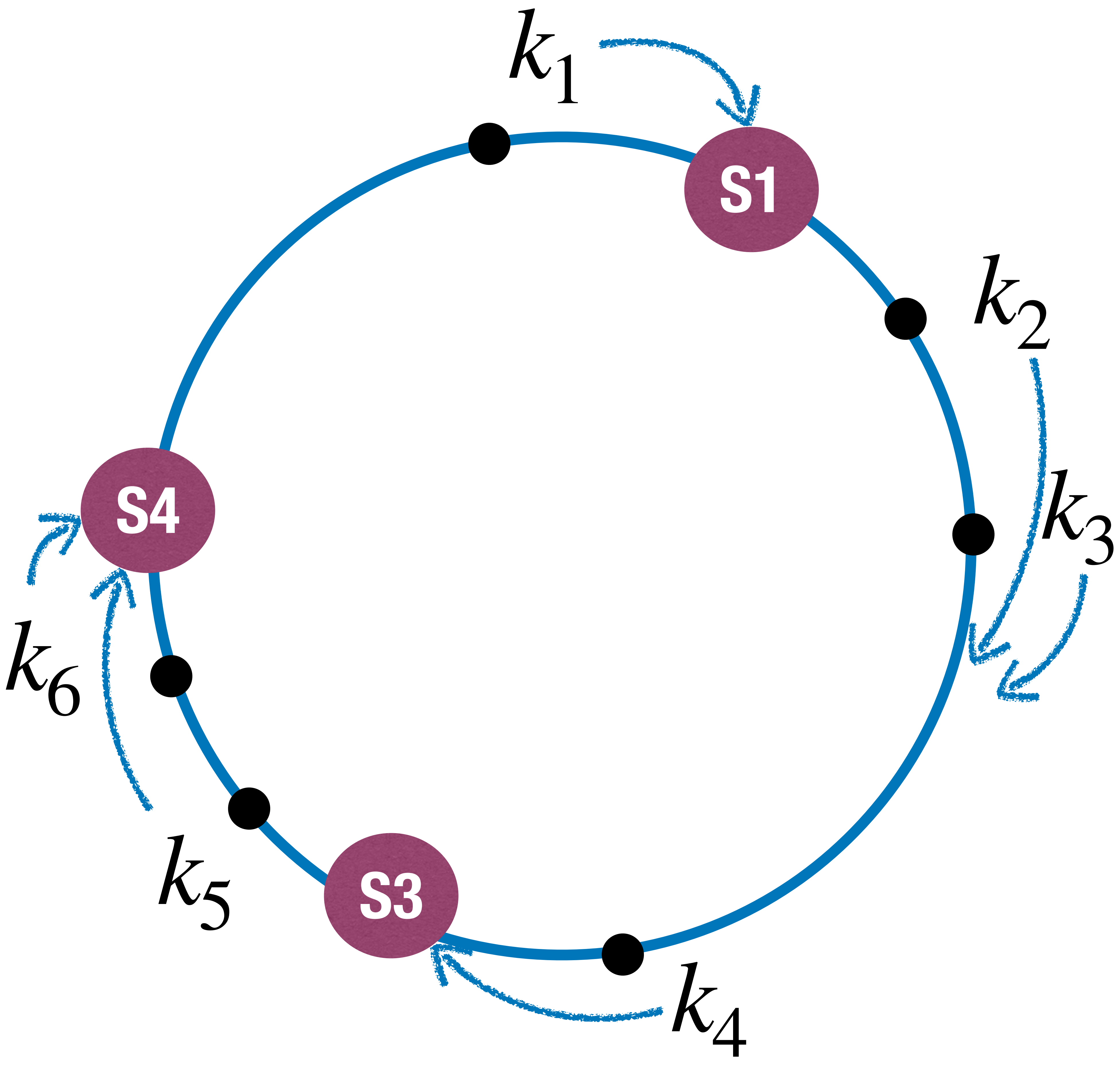
Consistent hashing

Remove a server (e.g., remove S2)



Consistent hashing

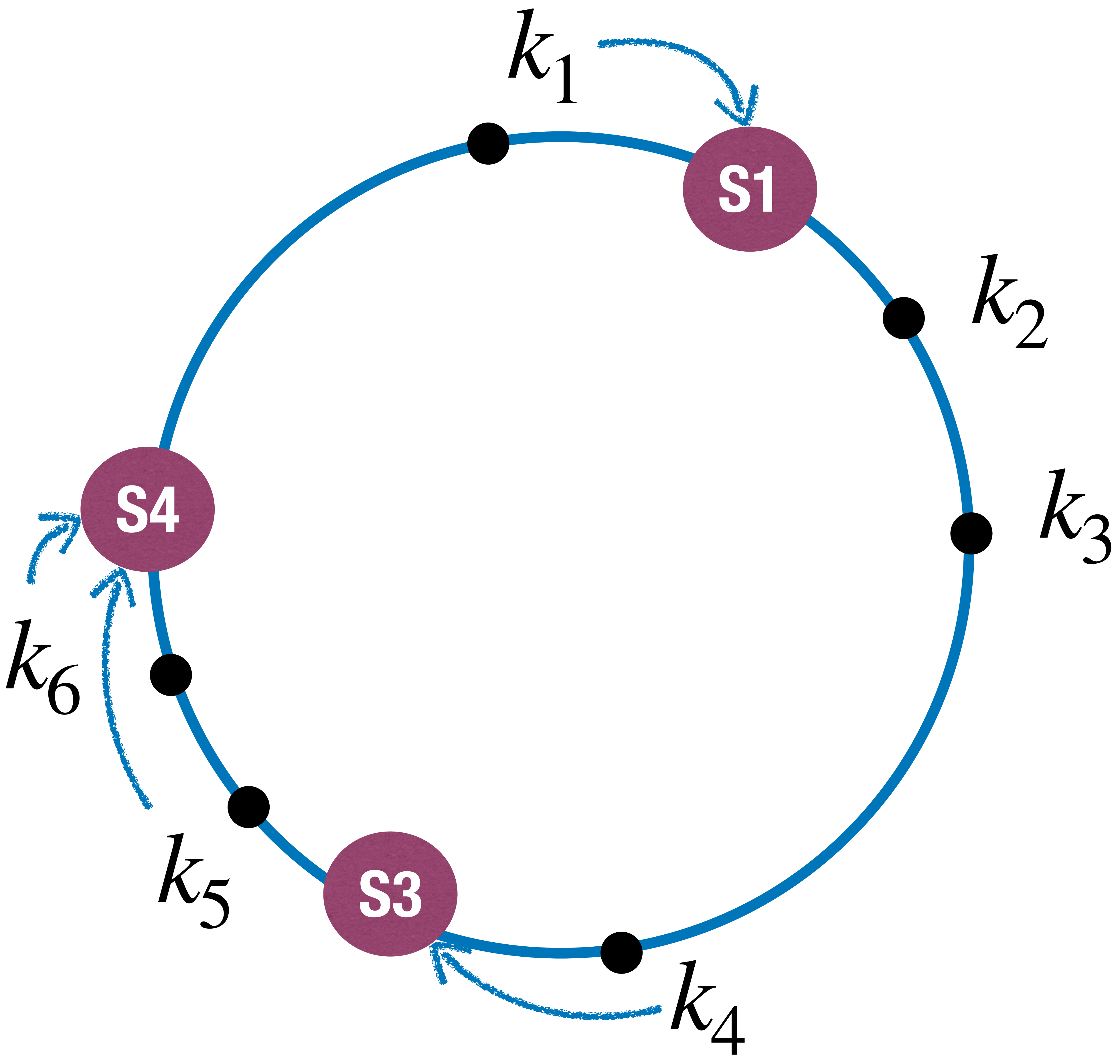
Remove a server (e.g., remove S2)



© 2025 Gengrui (Edward) Zhang

Consistent hashing

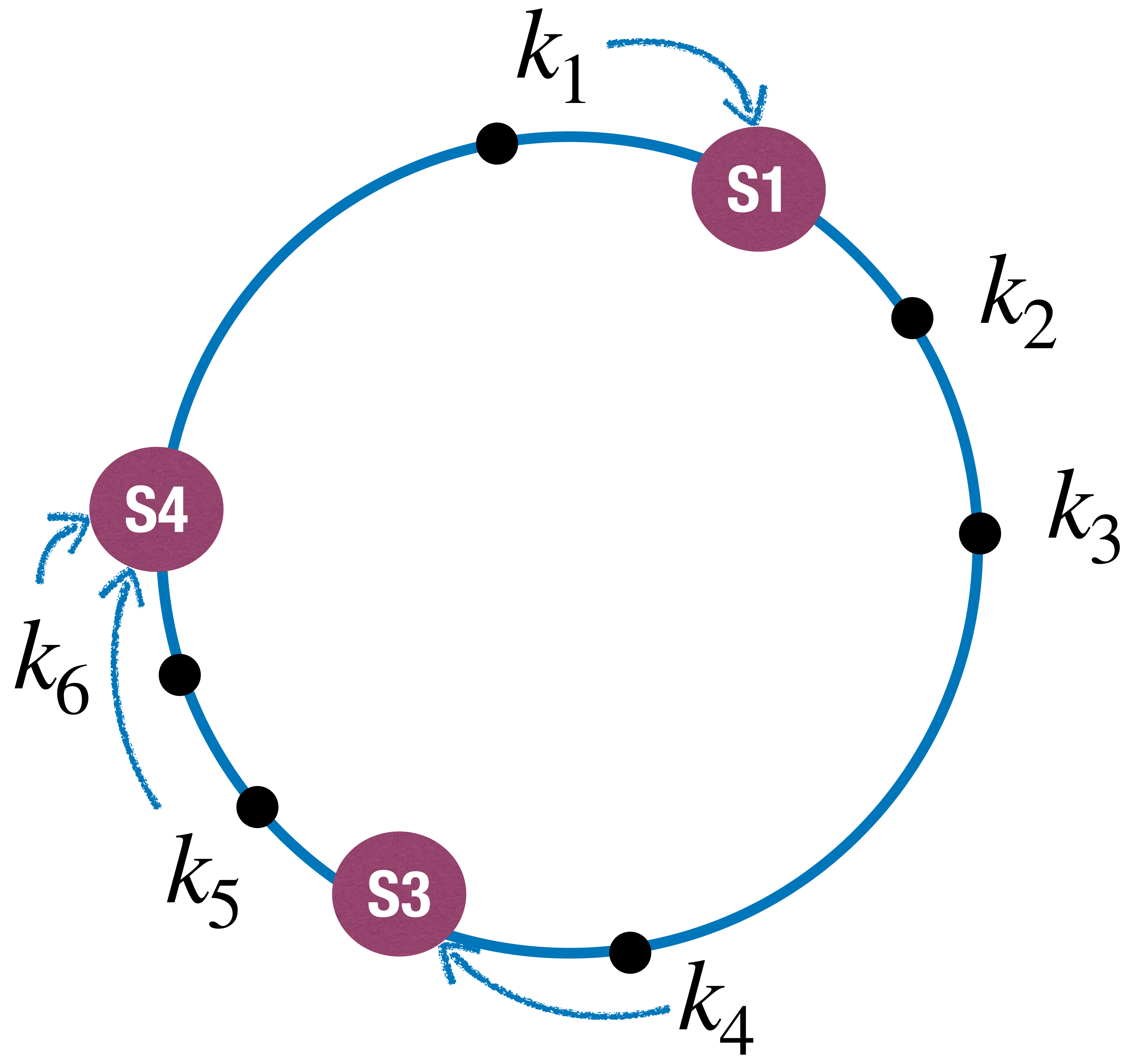
Remove a server (e.g., remove S2)



Consistent hashing

Remove a server (e.g., remove S2)

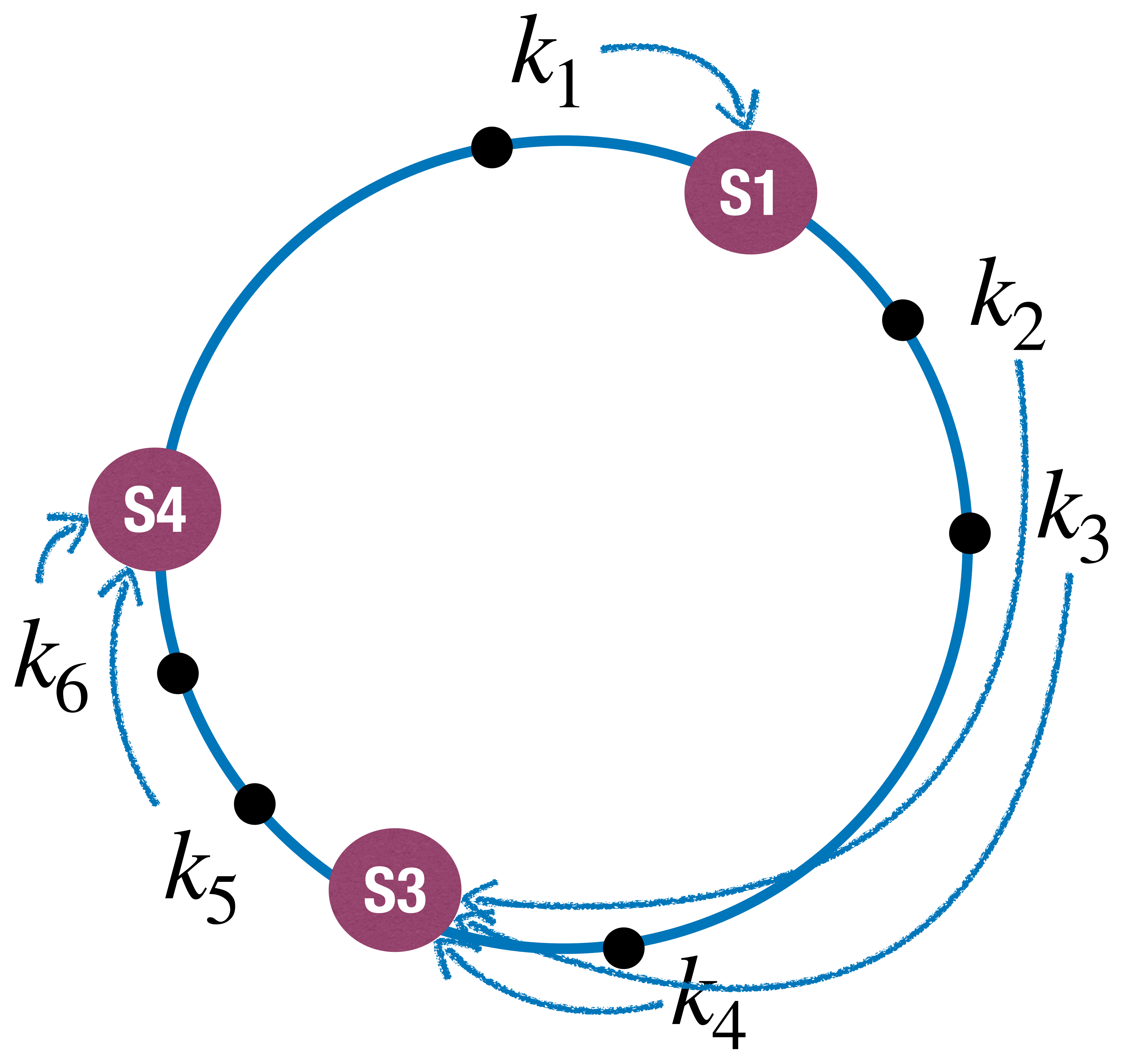
- Only keys in that interval are affected
- Move to the next clockwise server



Consistent hashing

Remove a server (e.g., remove S2)

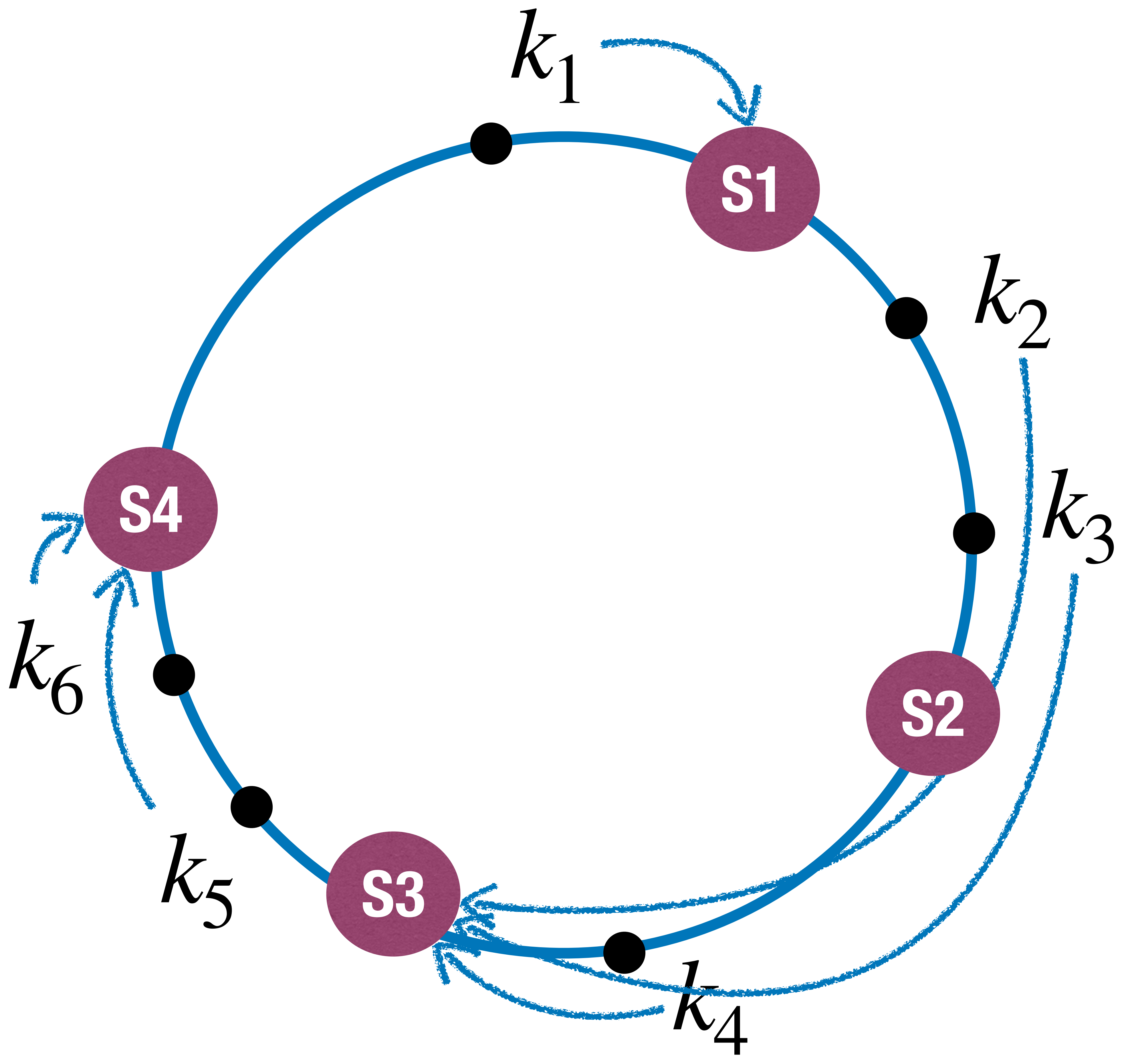
- Only keys in that interval are affected
- Move to the next clockwise server



Consistent hashing

Add a server

- E.g., Add Server 2 back
- Only the portion between S1 and S2 are affected
- Other keys are not redistributed

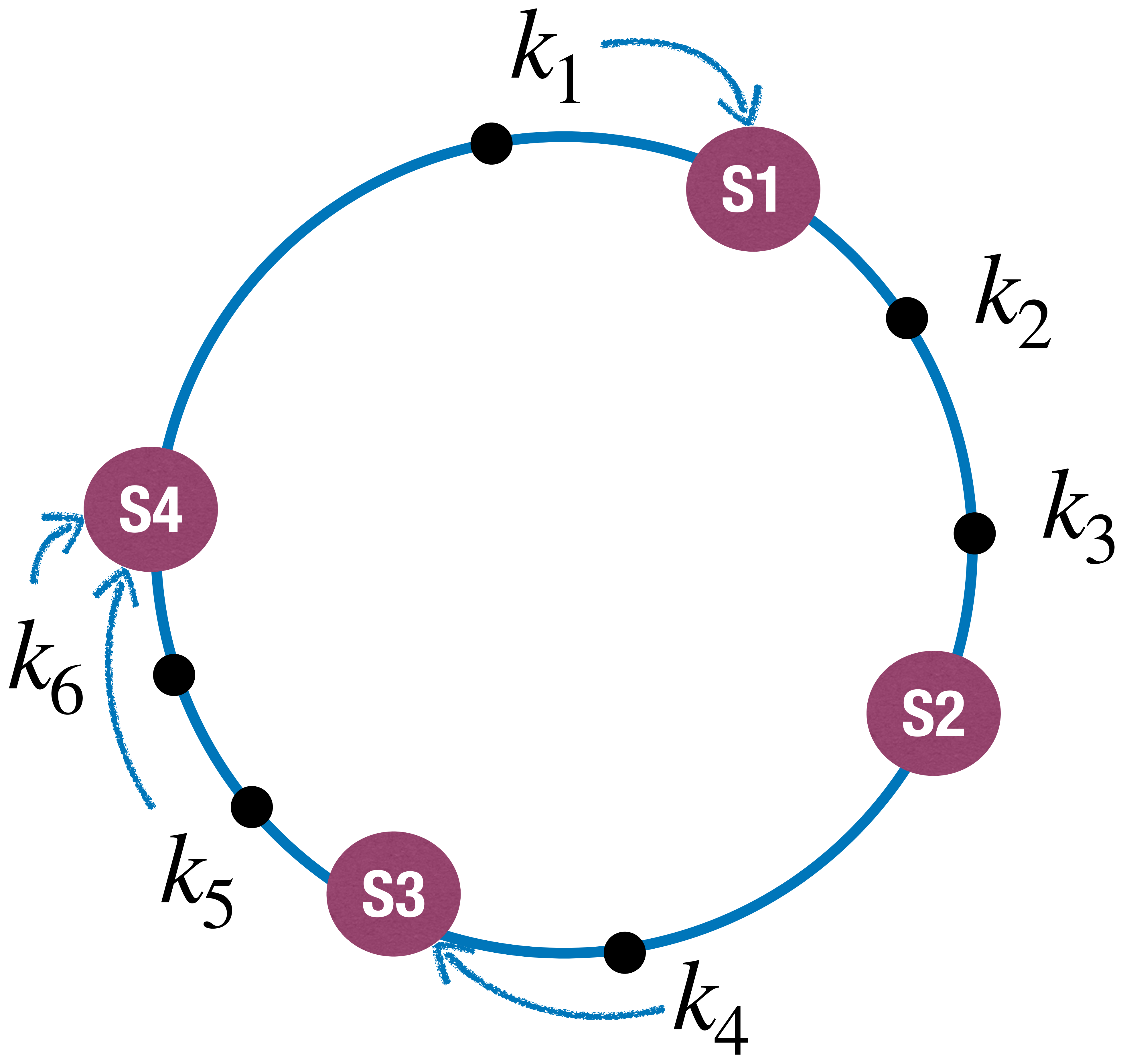


© 2025 Gengrui (Edward) Zhang

Consistent hashing

Add a server

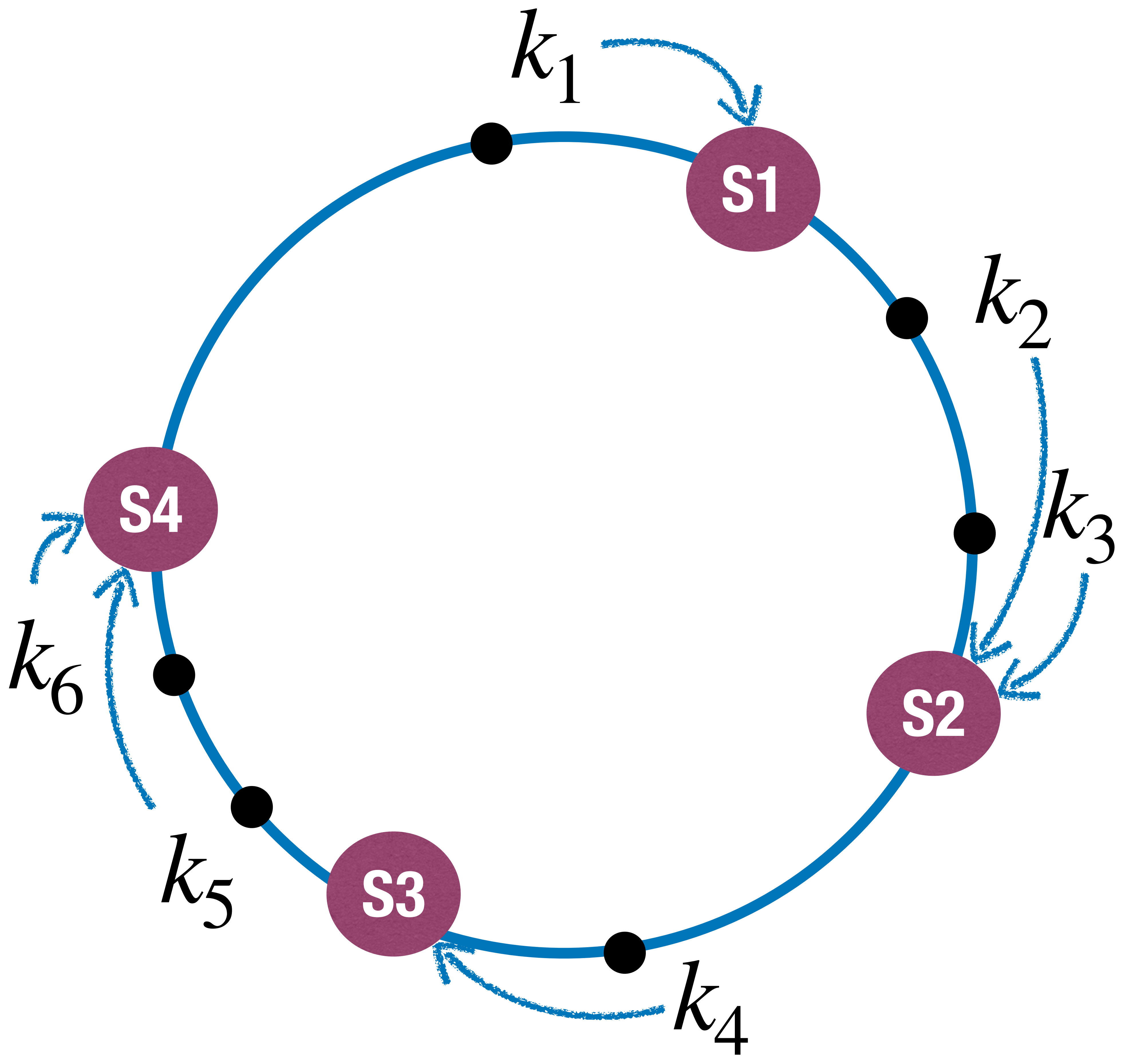
- E.g., Add Server 2 back
- Only the portion between S1 and S2 are affected
- Other keys are not redistributed



Consistent hashing

Add a server

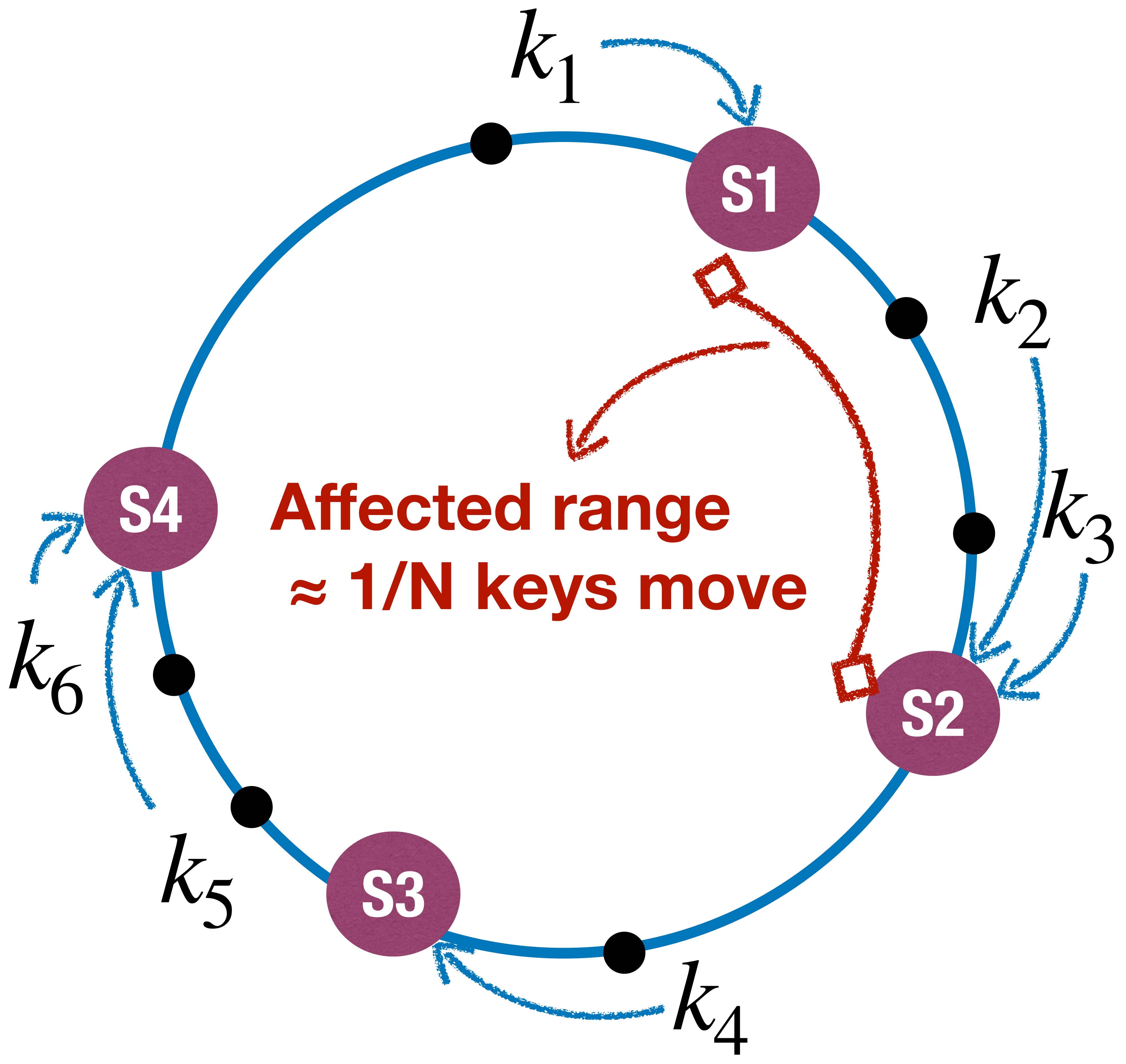
- E.g., Add Server 2 back
- Only the portion between S1 and S2 are affected
- Other keys are not redistributed



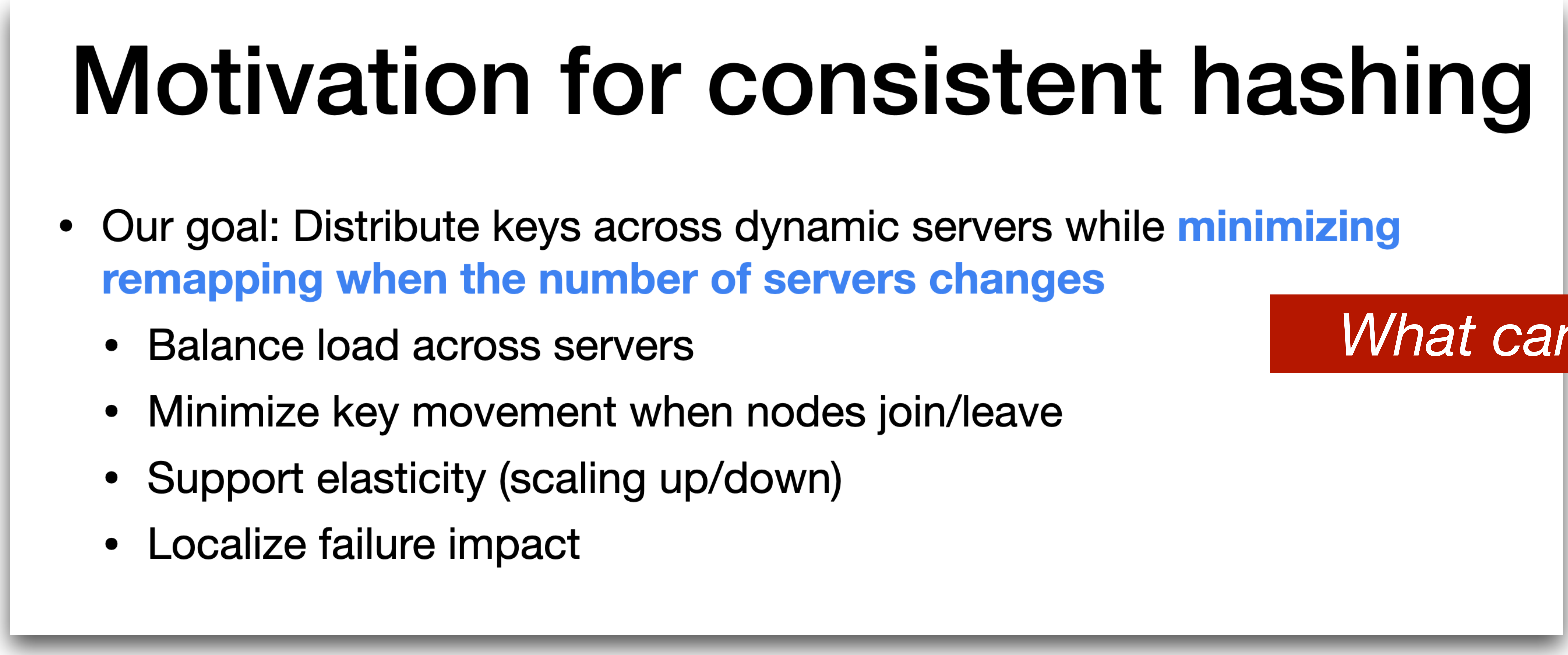
Consistent hashing

Add a server

- E.g., Add Server 2 back
- Only the portion between S1 and S2 are affected
- Other keys are not redistributed



Recall our goal. Is it perfect?

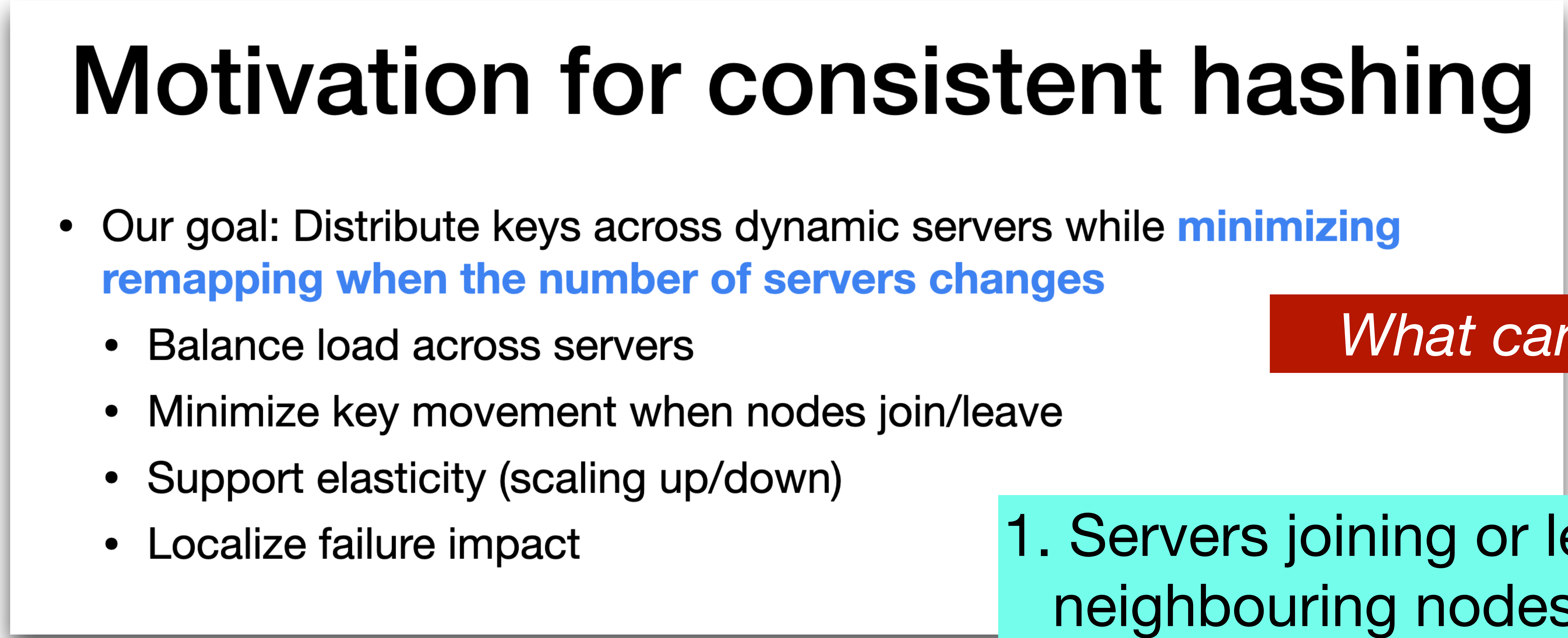


Motivation for consistent hashing

- Our goal: Distribute keys across dynamic servers while **minimizing remapping when the number of servers changes**
  - Balance load across servers
  - Minimize key movement when nodes join/leave
  - Support elasticity (scaling up/down)
  - Localize failure impact

What can go wrong ?

Recall our goal. Is it perfect?



Motivation for consistent hashing

- Our goal: Distribute keys across dynamic servers while **minimizing remapping when the number of servers changes**
  - Balance load across servers
  - Minimize key movement when nodes join/leave
  - Support elasticity (scaling up/down)
  - Localize failure impact

What can go wrong ?

1. Servers joining or leaving shift loads to neighbouring nodes
2. Workload characteristic may not uniformly distribute the load across nodes

Recall memory management in OS

- Recall how one of the key concepts in Operating Systems
  - How do we manage memory?

Memory without virtualization

- One contiguous block of physical memory
- No paging
- Leading to fragmentation

Recall memory management in OS

- Recall how one of the key concepts in Operating Systems
  - How do we manage memory?

Memory without virtualization

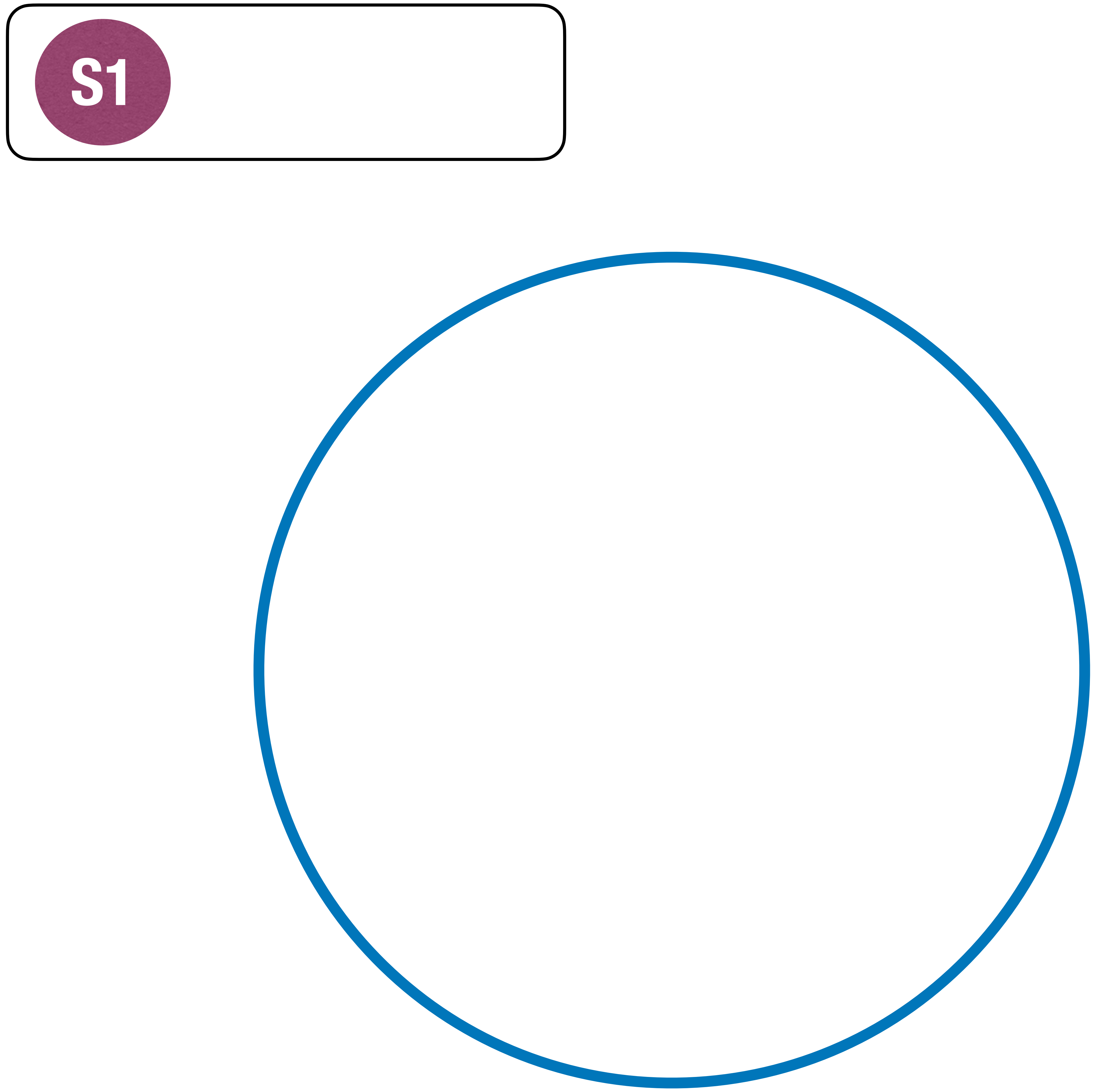
- One contiguous block of physical memory
- No paging
- Leading to fragmentation

Adding virtual memory

- Break memory into fixed-size pages
- Create virtual pages
- Map virtual to physical frames via page table
- Fine-grained load distribution

Similarly, adding virtual nodes to hash ring

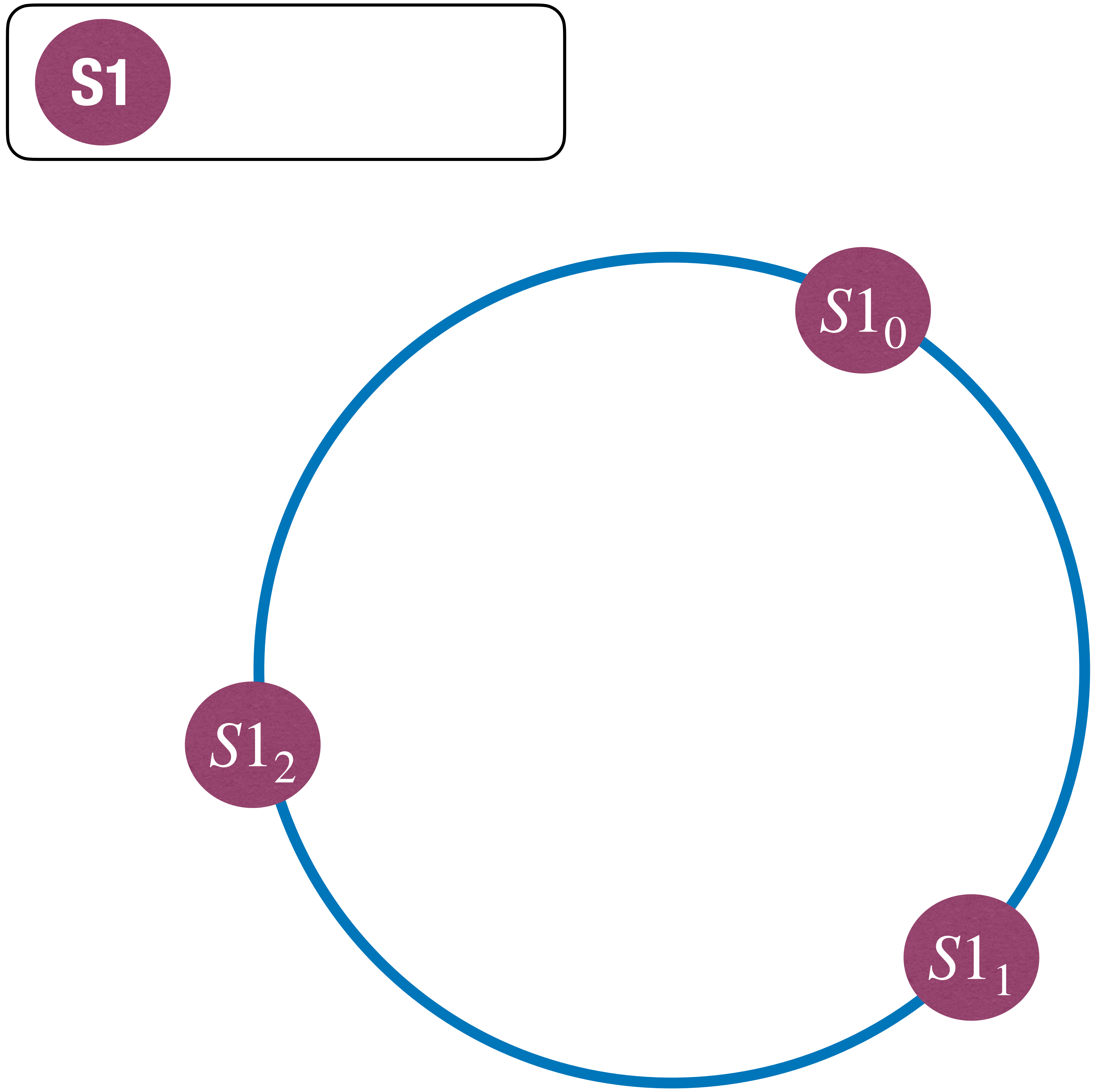
- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



© 2025 Gengrui (Edward) Zhang

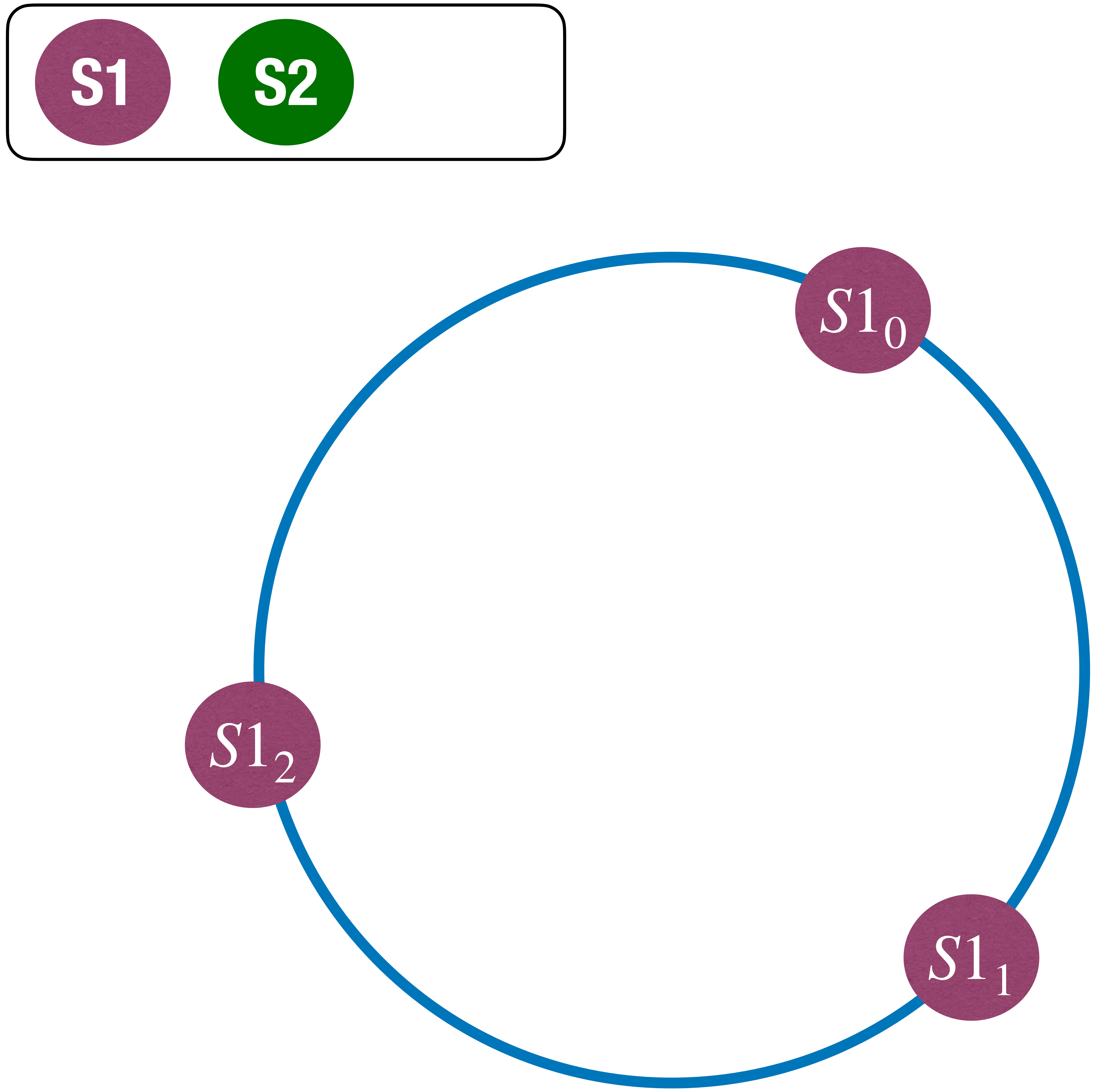
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



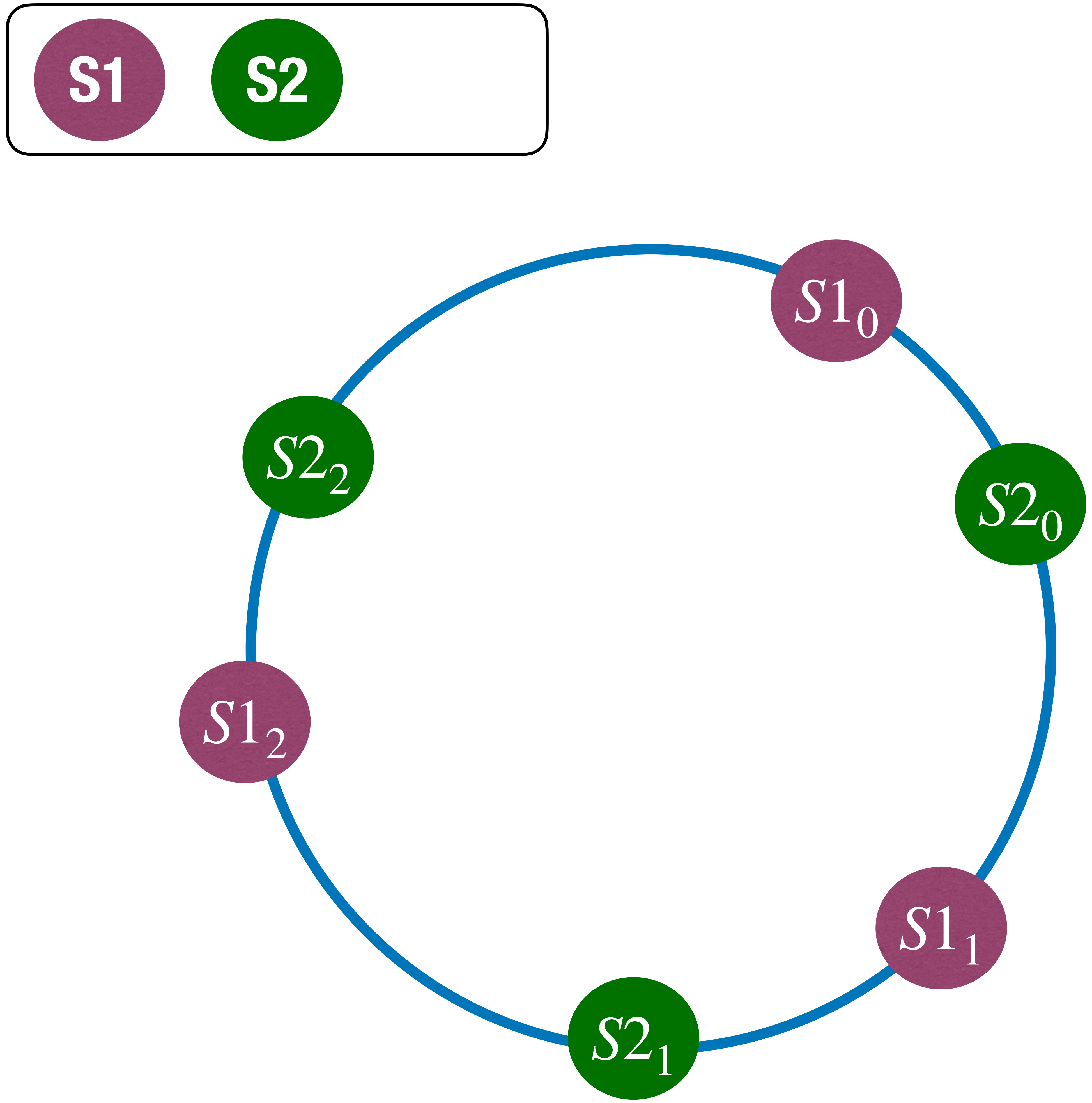
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



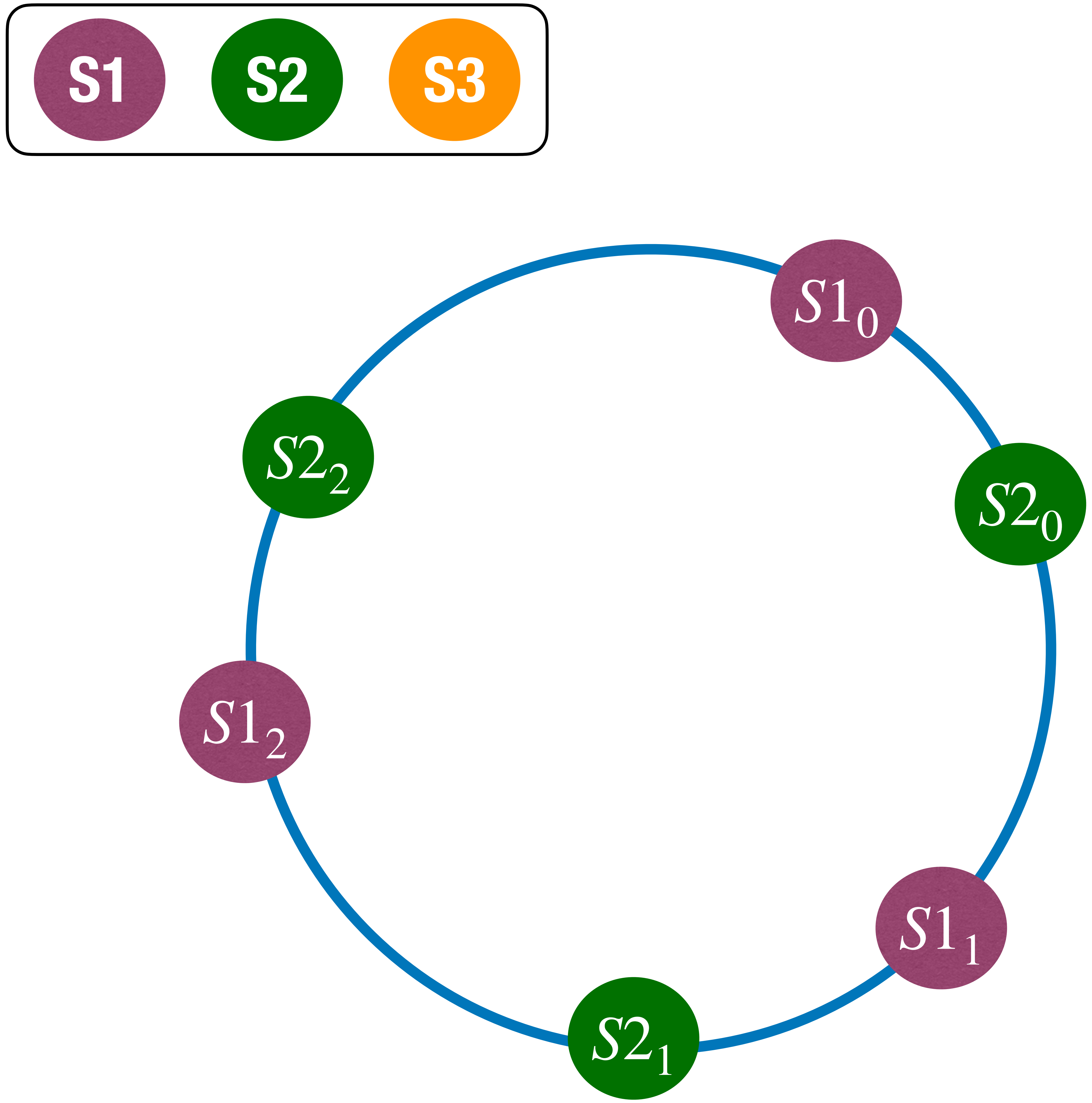
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



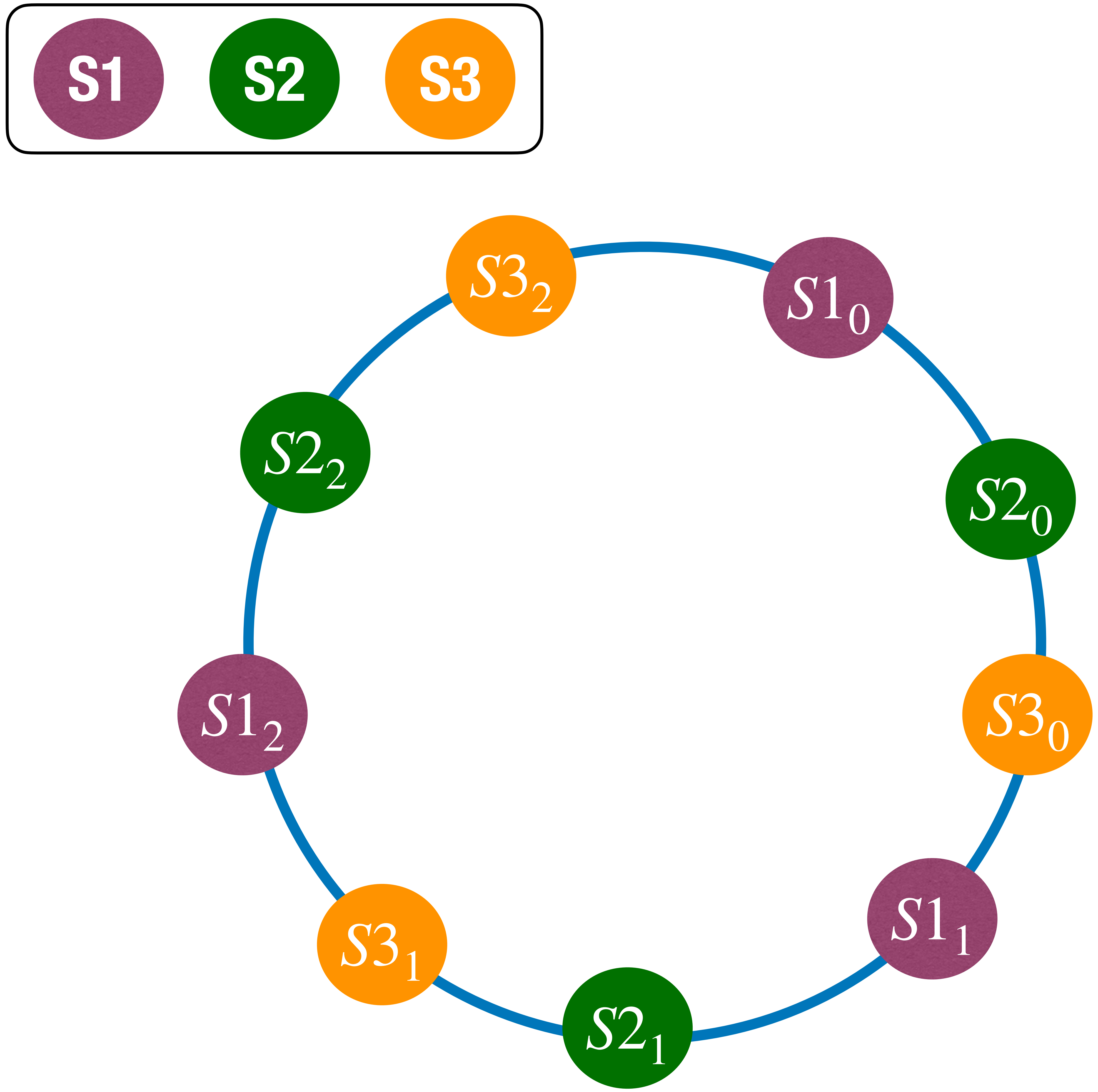
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



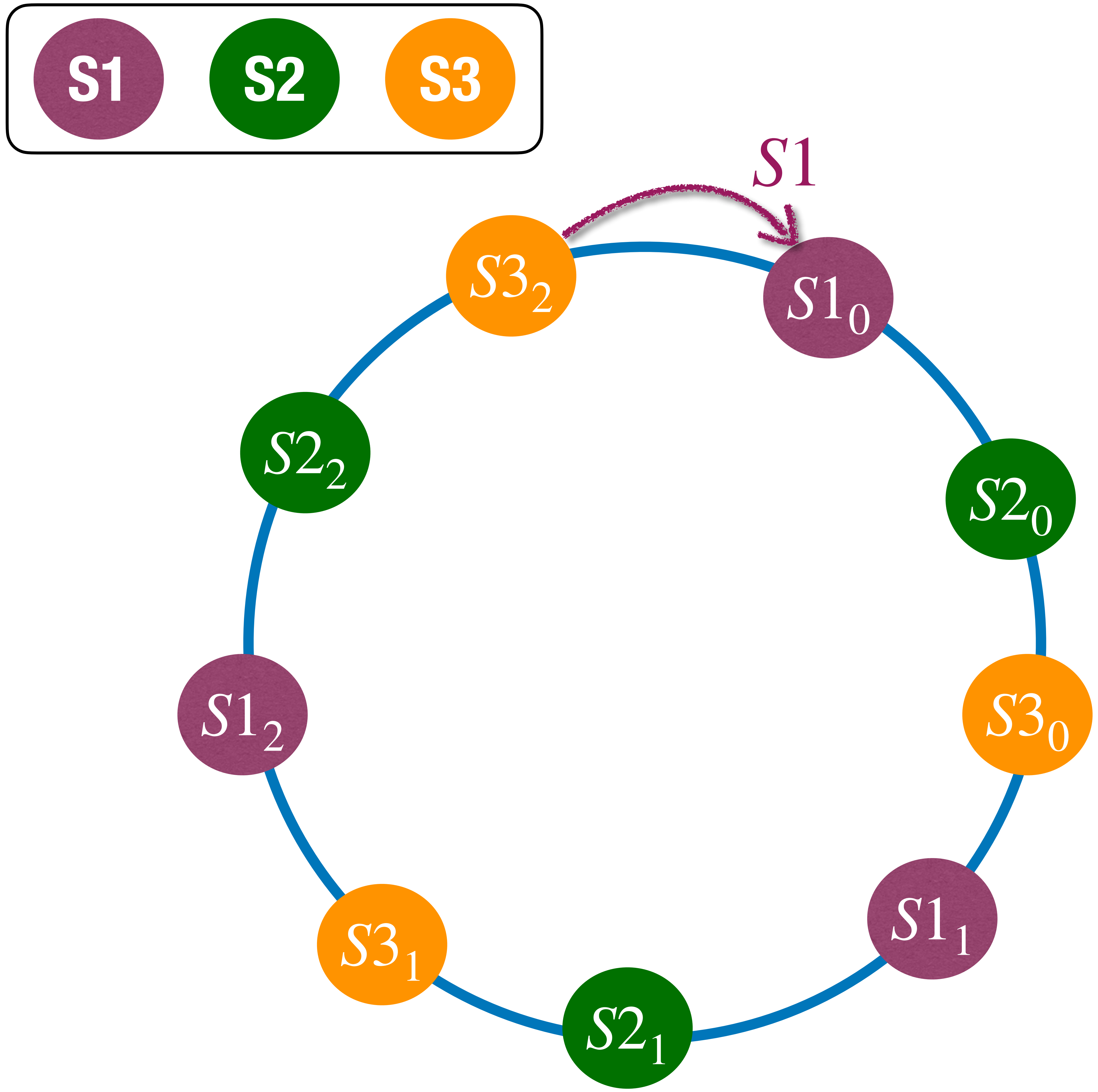
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



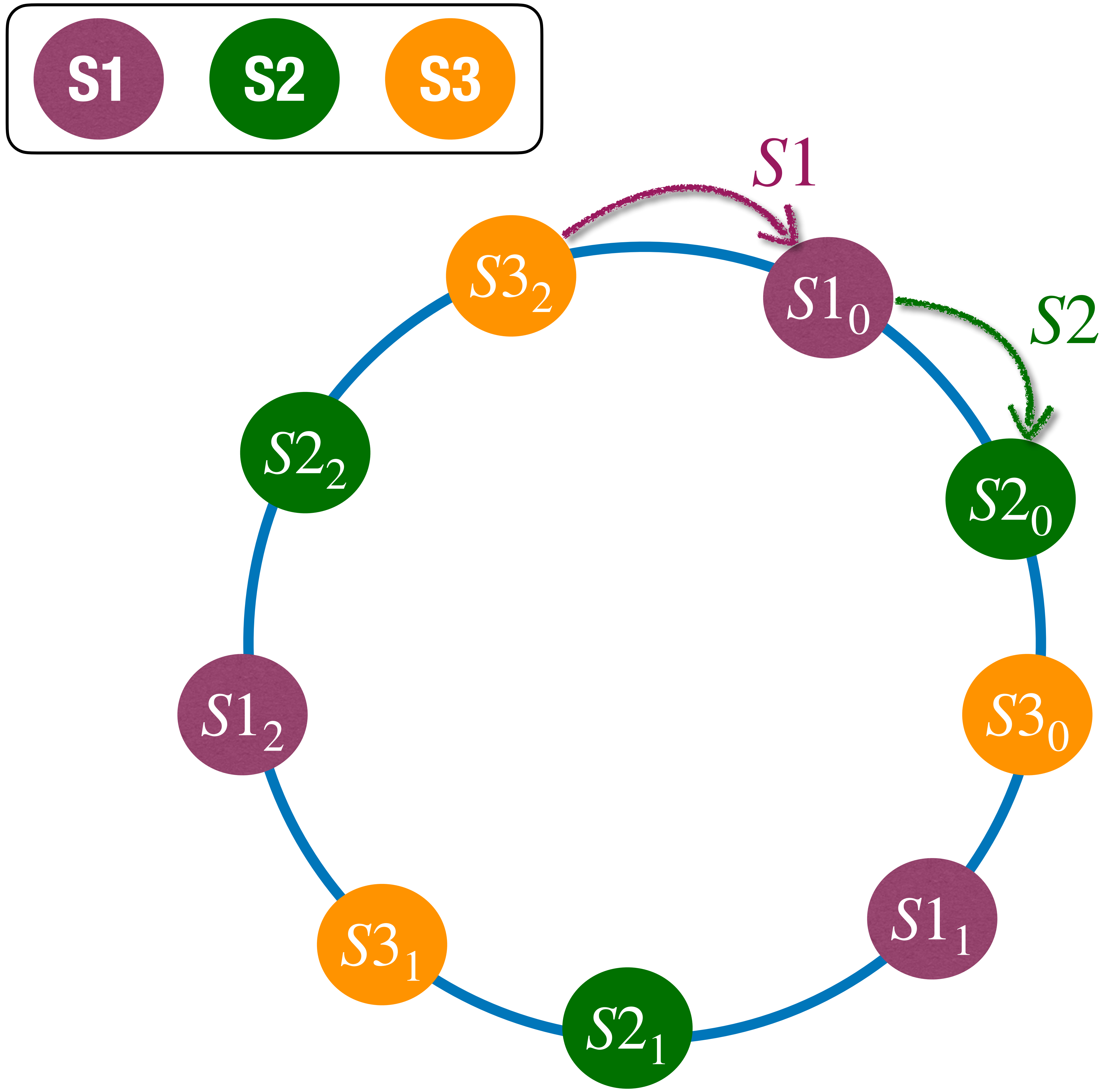
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



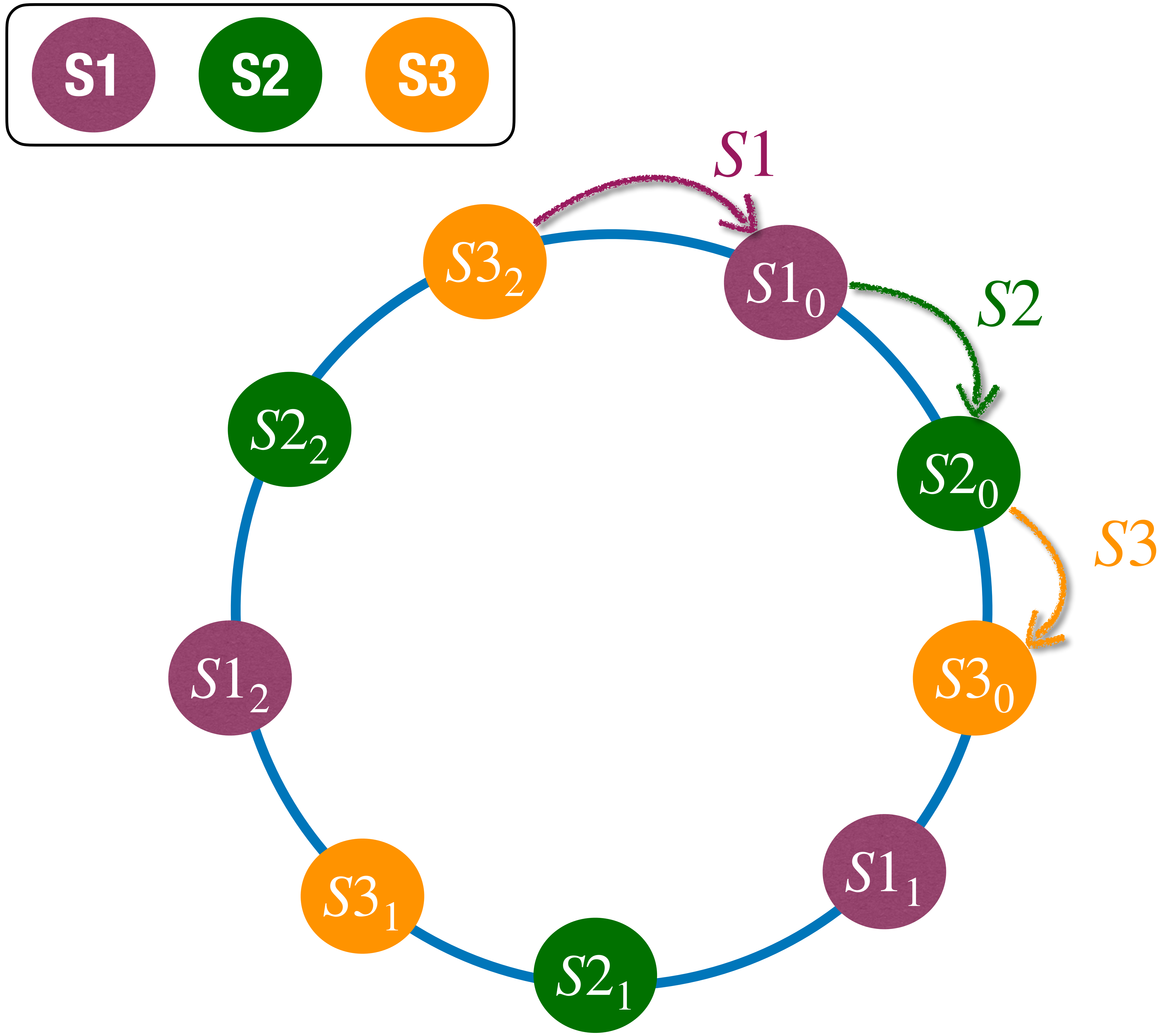
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



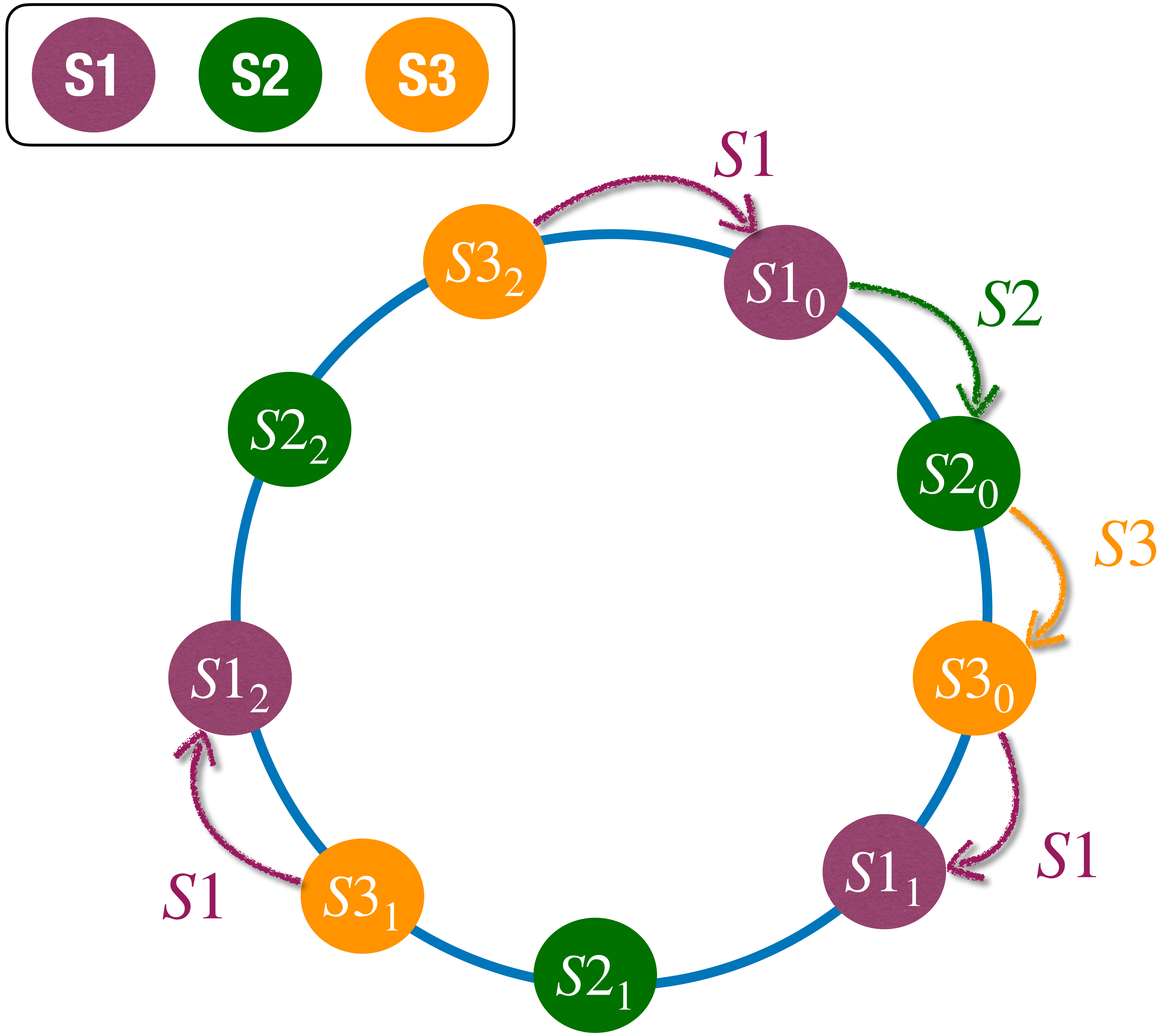
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



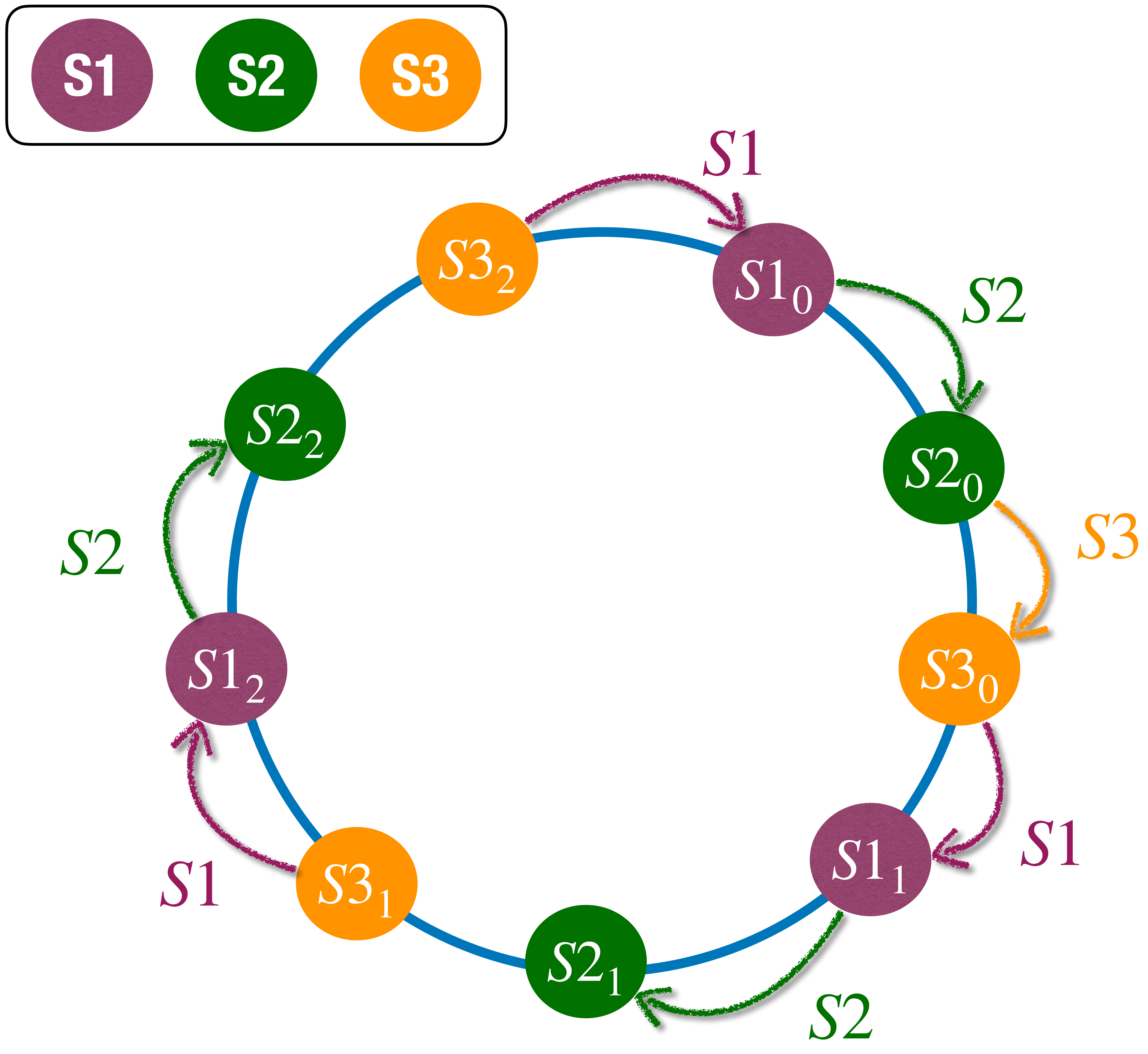
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



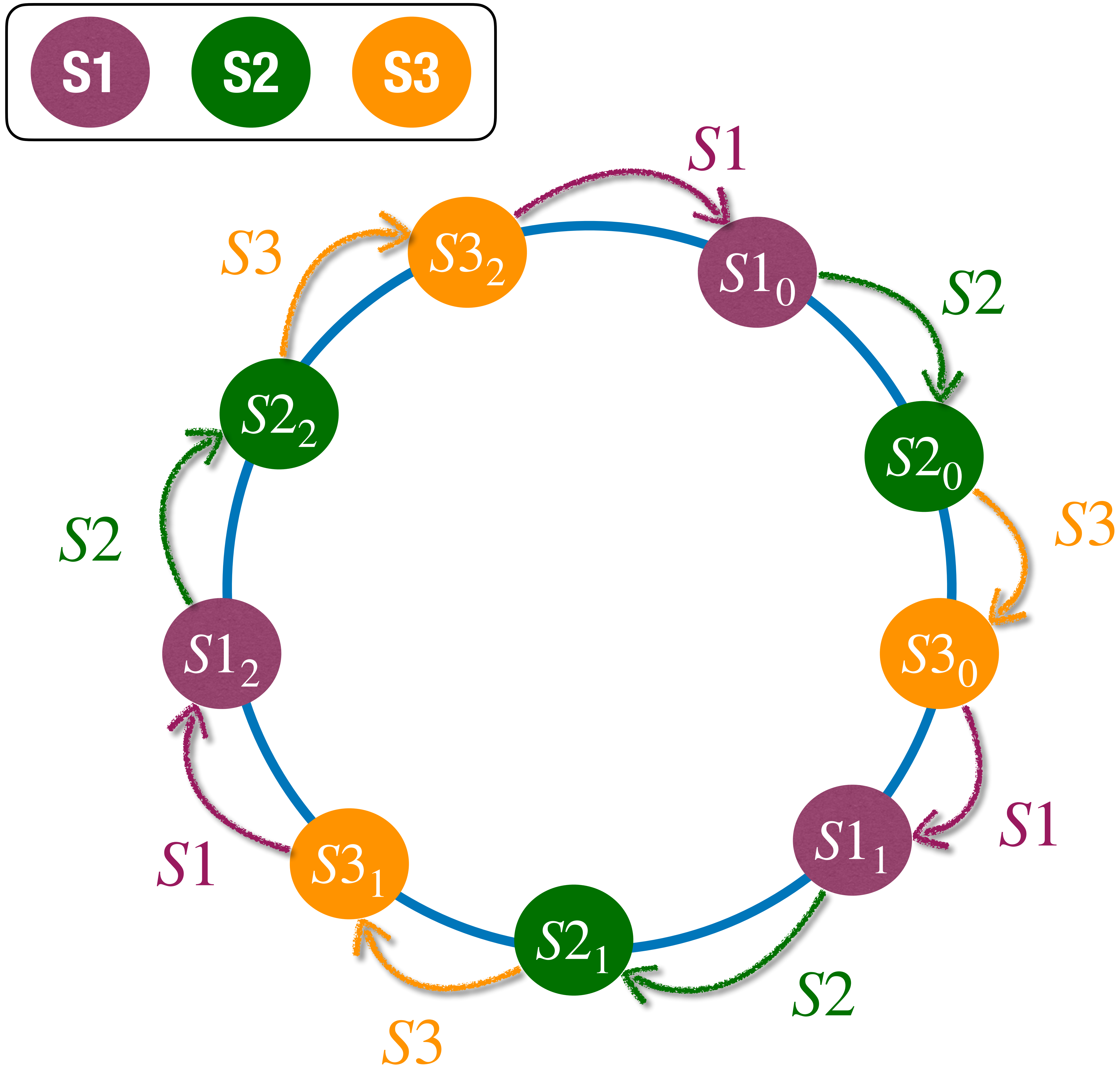
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



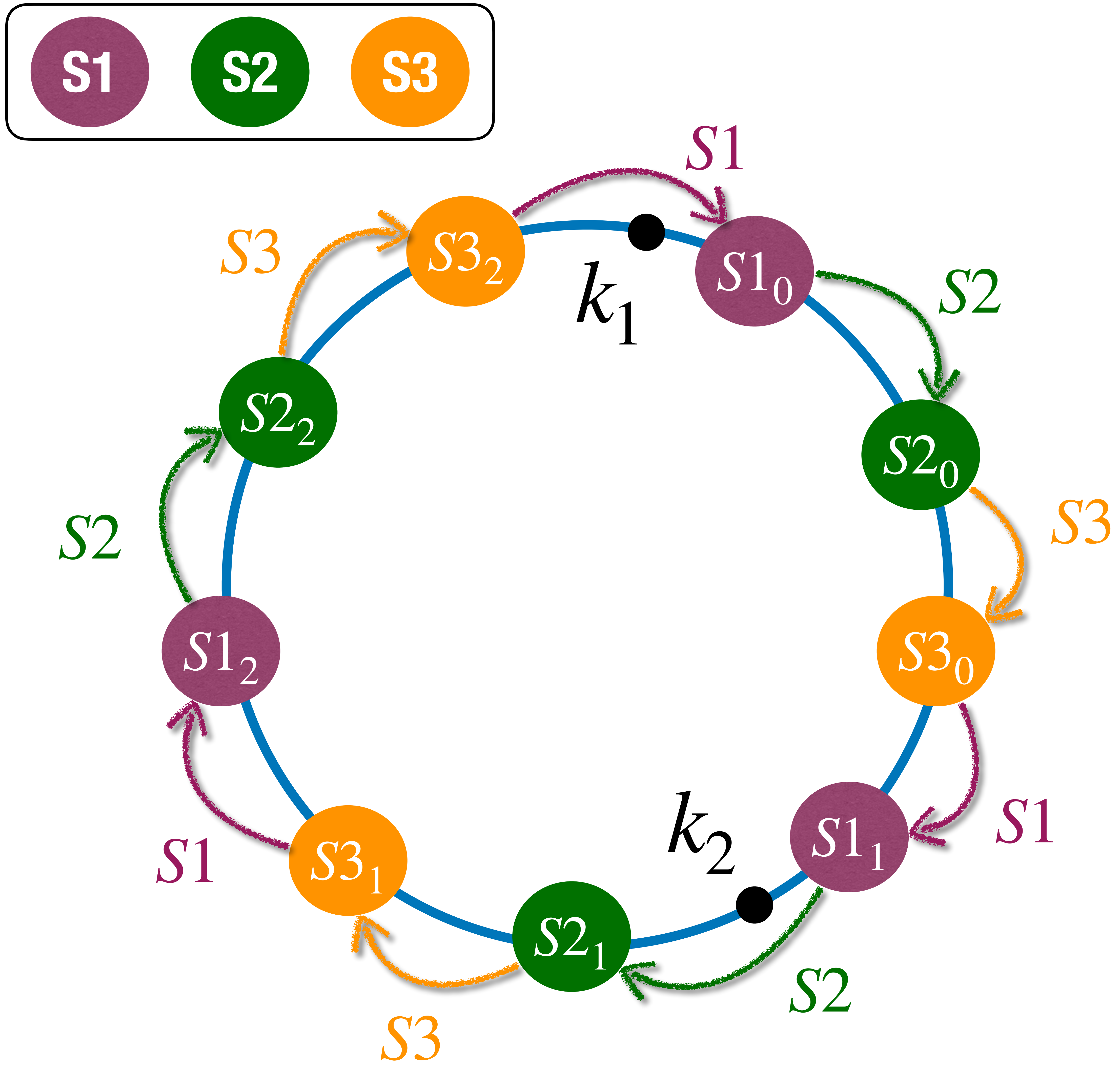
Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



Similarly, adding virtual nodes to hash ring

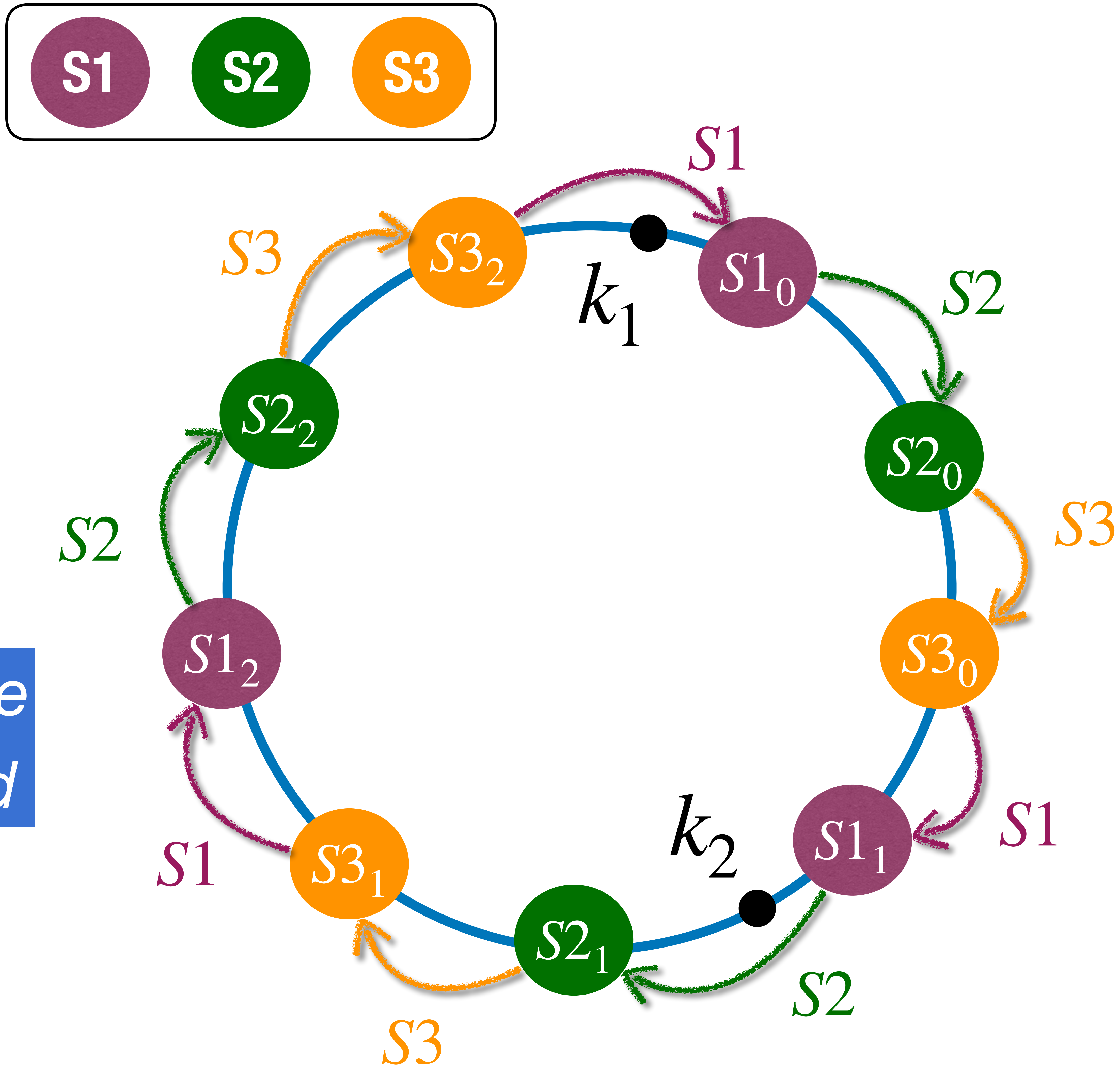
- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$



Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$

As the number of virtual nodes increases, the distribution of keys becomes more balanced

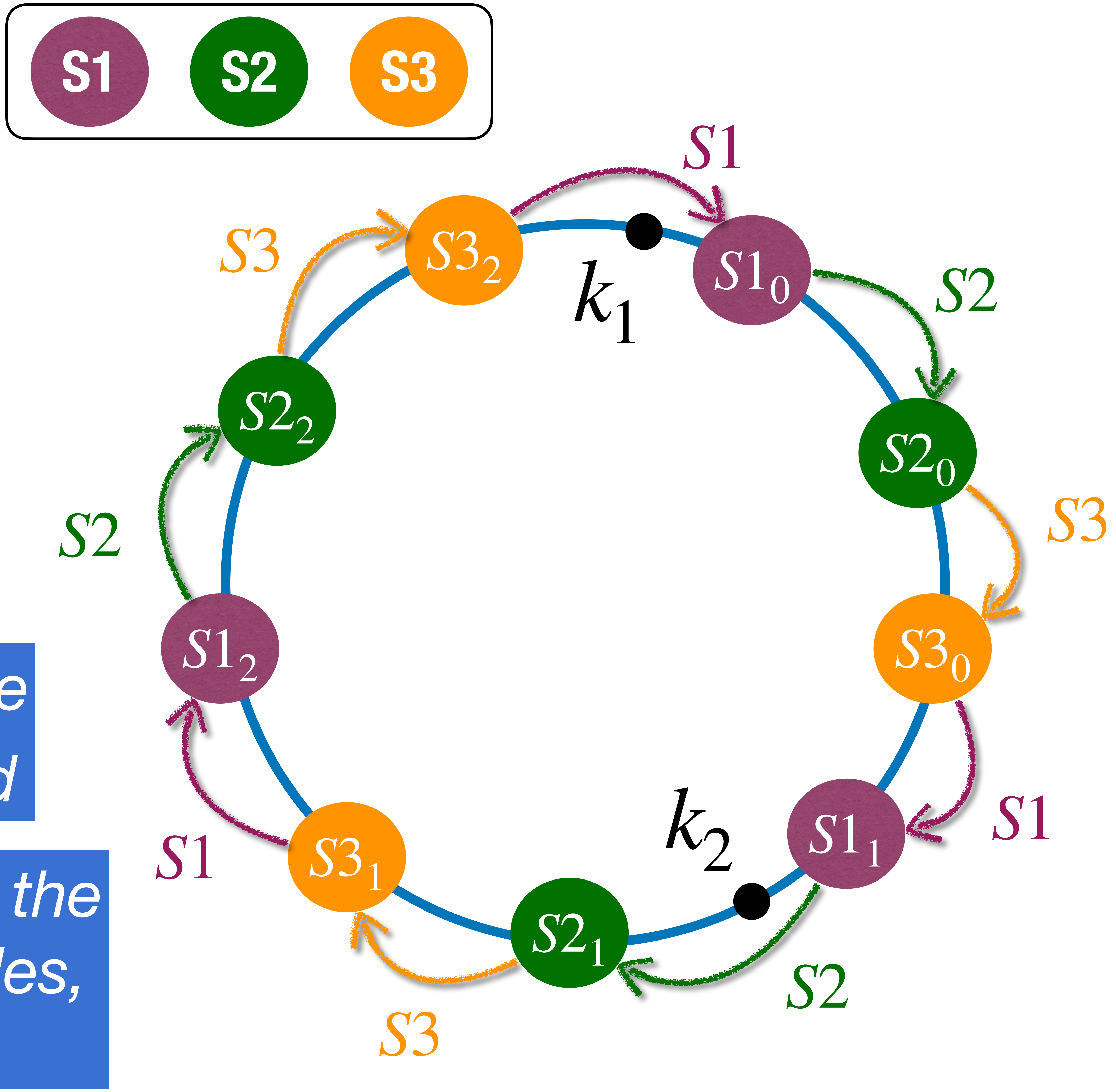


Similarly, adding virtual nodes to hash ring

- Virtual nodes (vnodes) are multiple logical positions on the hash ring that map to the same physical server.
  - N = # of physical servers
  - V = # of virtual nodes per server
  - Total positions on ring: $N \times V$

As the number of virtual nodes increases, the distribution of keys becomes more balanced

Support heterogeneous servers by adjusting the size of range. E.g., strong servers: 200 Vnodes, weak servers 50 Vnodes



Summary of consistent hashing

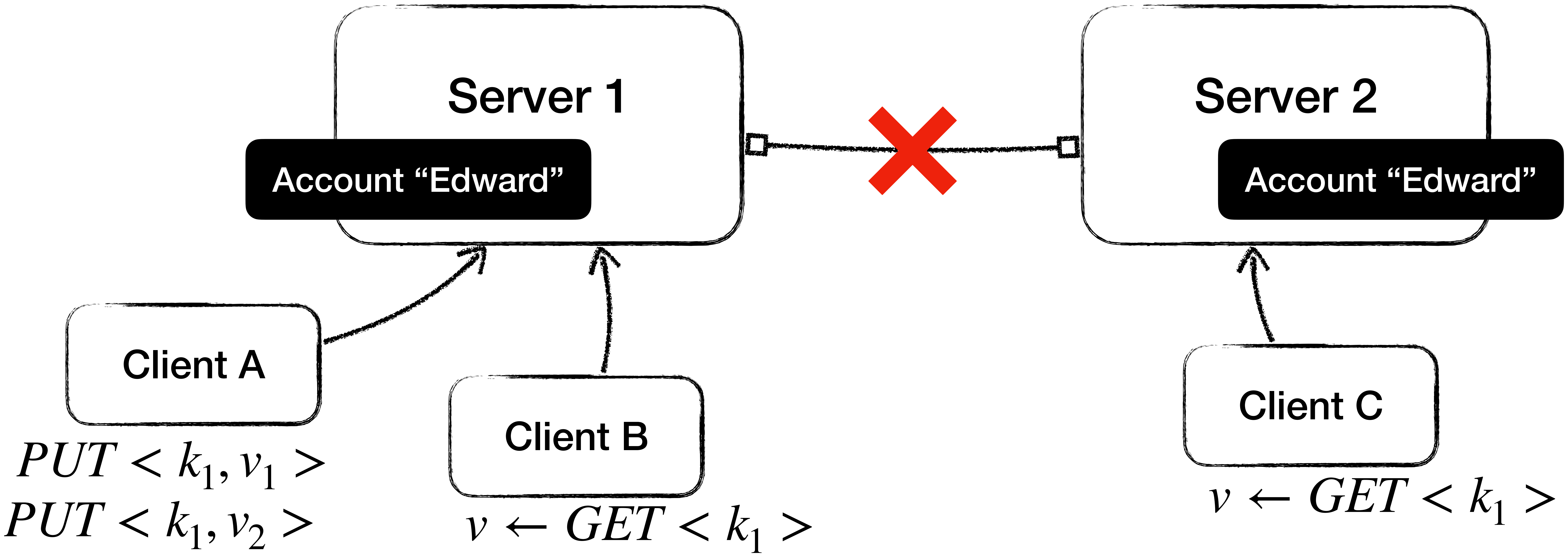
- Maps keys and servers into the same circular hash space (ring)
- Each key assigned to the next clockwise server
- When a server joins/leaves, only ≈ 1/N keys move
- Avoids global reshuffling required by naïve hashing
- Enables elastic scaling and graceful failure handling
- Uses virtual nodes to improve load balance and support heterogeneous servers

Today’s outline

Consistent Hashing

CAP

A simple network partition

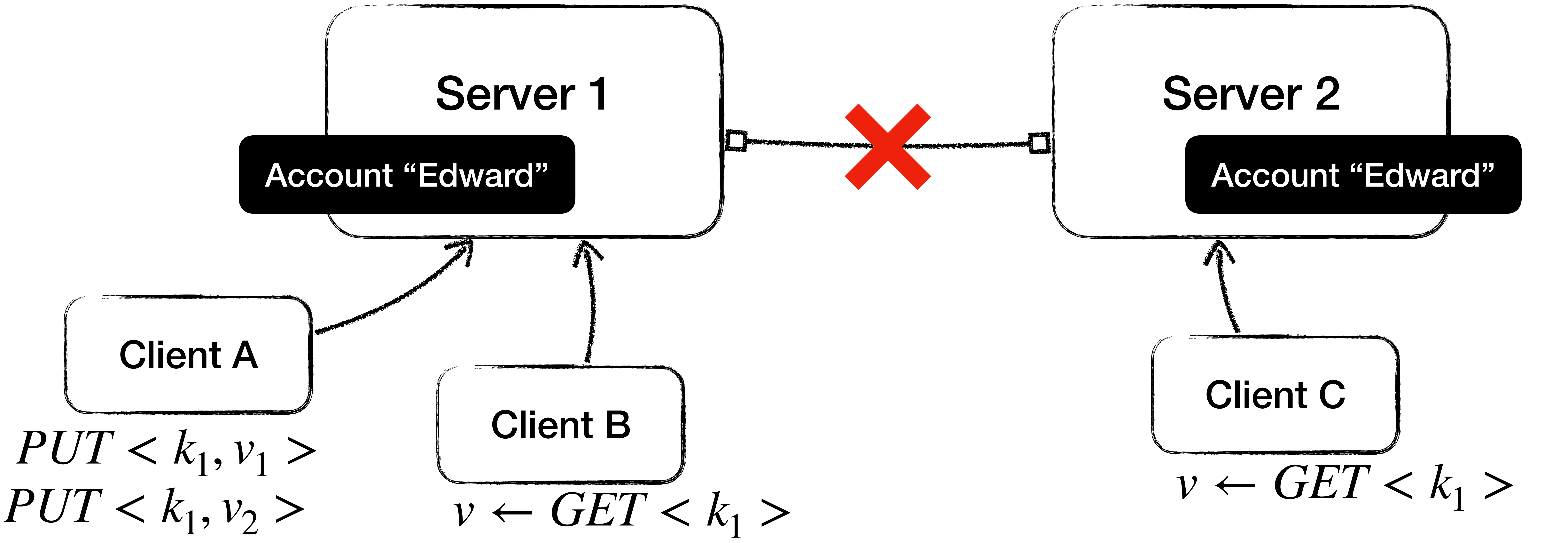


$PUT < k_1, v_1 >$
$PUT < k_1, v_2 >$

$v \leftarrow GET < k_1 >$

$v \leftarrow GET < k_1 >$

A simple network partition



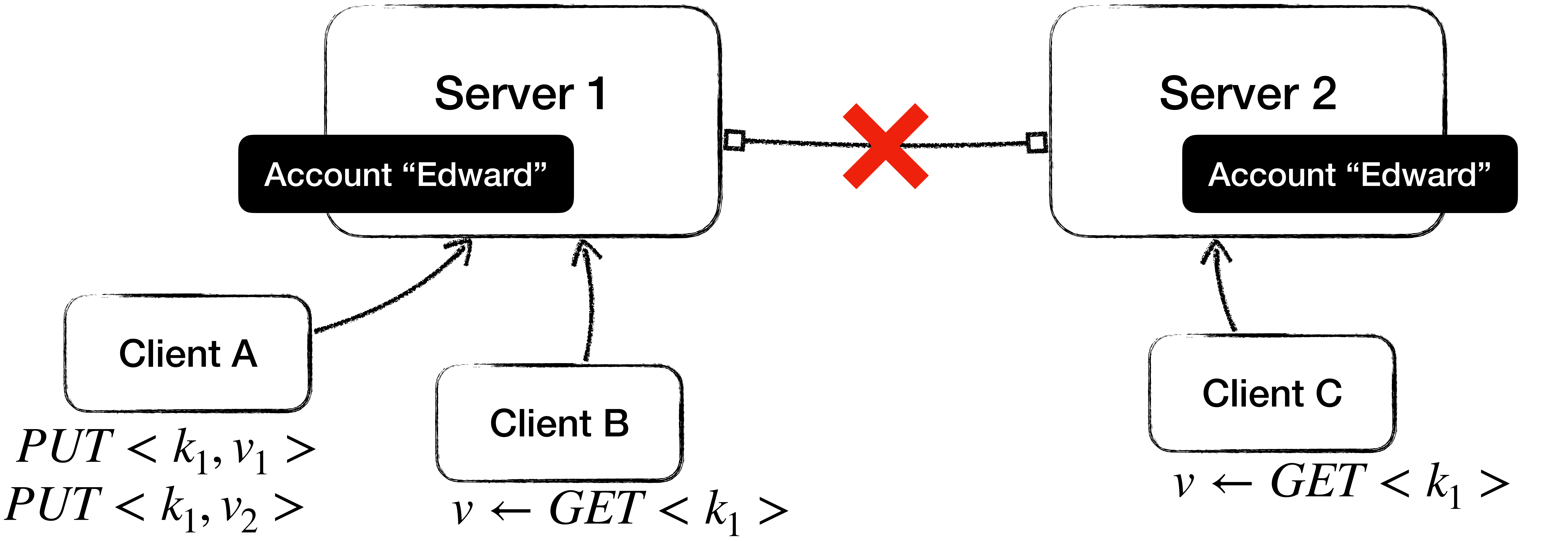
$PUT < k_1, v_1 >$
$PUT < k_1, v_2 >$

$v \leftarrow GET < k_1 >$

$v \leftarrow GET < k_1 >$

- Say it's a banking application

A simple network partition



$PUT < k_1, v_1 >$

$PUT < k_1, v_2 >$

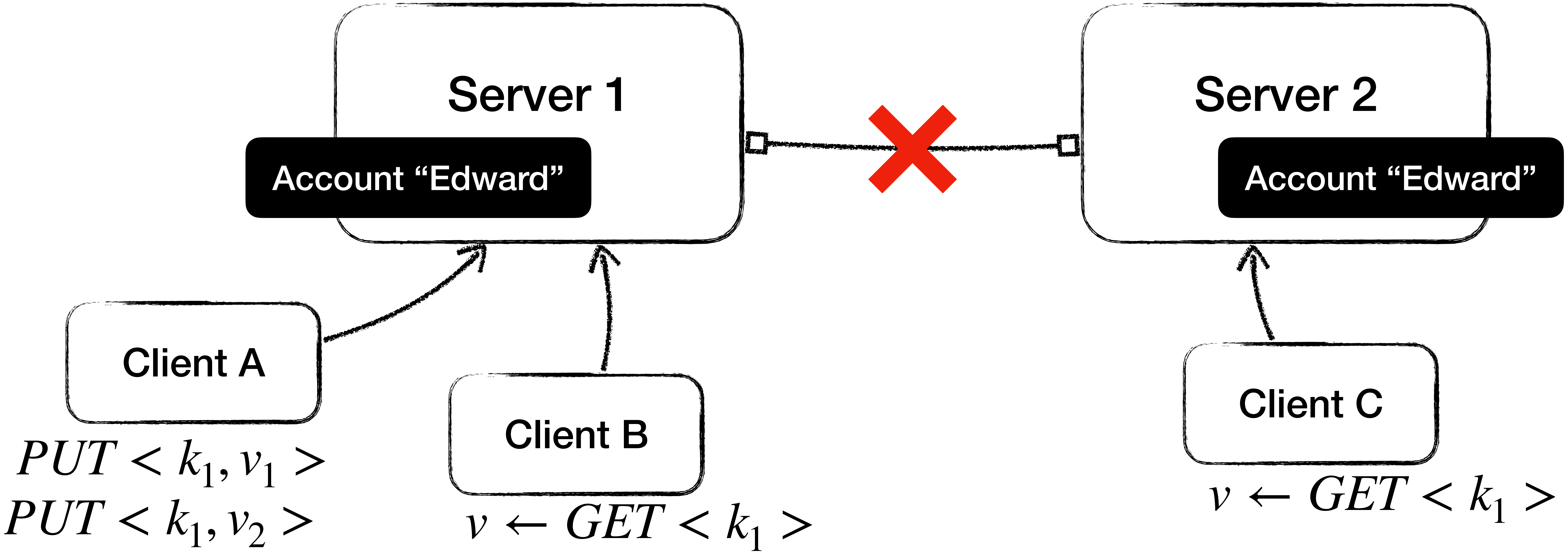
$v \leftarrow GET < k_1 >$

$v \leftarrow GET < k_1 >$

- Say it’s a banking application

Consistency > Availability

A simple network partition



$PUT < k_1, v_1 >$
$PUT < k_1, v_2 >$

$v \leftarrow GET < k_1 >$

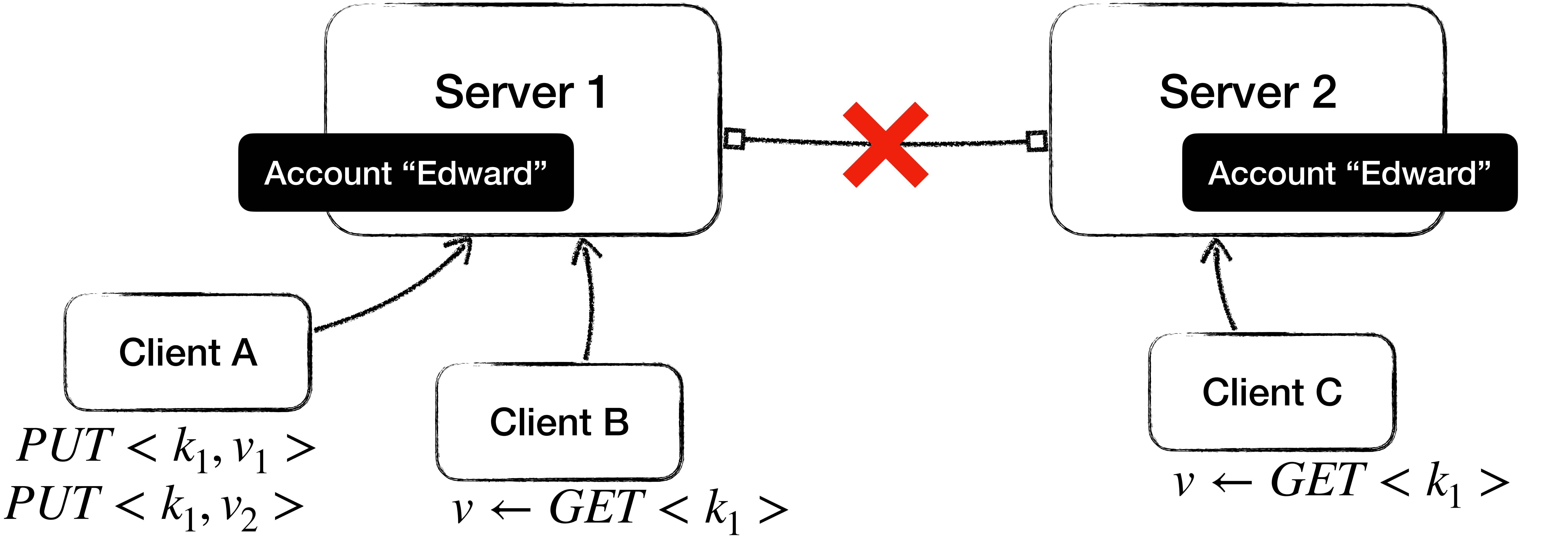
$v \leftarrow GET < k_1 >$

- Say it's a banking application

Consistency > Availability

- What if it's a social media application

A simple network partition



$PUT < k_1, v_1 >$
$PUT < k_1, v_2 >$

$v \leftarrow GET < k_1 >$

$v \leftarrow GET < k_1 >$

- Say it's a banking application

Consistency > Availability

- What if it's a social media application

Availability > Consistency

The CAP theorem

- It is **impossible** for a replicated **read-write store** in an asynchronous network to maintain the following **three guarantees simultaneously**:
    - Consistency
    - Availability
    - Partition-Tolerance
- Initially, conjectured by Eric Brewer in 1998, later proven by Lynch et al.
- Describes **tradeoffs** involved in distributed system design

CAP

- **Consistency:**
  - All read requests should read the latest value (or return an error)
- **Availability:**
  - All requests should return successfully
- **Partition-tolerance:**
  - The system can tolerate arbitrary number of communication failures
- Traditional view
  - Today, more a spectrum

Definition of Consistency

- Refers to replication consistency
  - Not related to A-C-I-D properties for transactions
- Ideally means strict consistency
  - As we know, this is by and large impossible in a distributed system
- Thus, here, assumes linearizability
- This usually means replication across sites should be done eagerly

Definition of Availability

- Every request received by a non-failed node must result in a non-error response
    - Non-triviality requirement: a system which always responds with errors is not available
- Assumes a crash failure model for nodes
    - Functioning nodes must continue to operate even if there are failed nodes in system
- No requirement on latency: response can be very slow but must eventually come through
- Both a weak and strong definition: no latency guarantee, but 100% response success

Definition of Partition-Tolerance

- Asynchronous system model
- Message loss (failure model)
- Partition means total communication loss between partitioned subsystems
- System continues request processing even if a network partition causes communication loss between subsystems
- If the system requires a stronger system model, or a weaker failure model, then it is not partition-tolerant
- No guarantee that partitions recover, but it doesn’t mean they are always present either

Common misunderstanding in CAP

- CAP does NOT say:
    - You can only pick two and permanently give up the third
- It says:
    - When a partition happens, you must choose between C and A

CP vs AP

- In advanced distributed systems (e.g., blockchains, cloud DBMS):
  - CP systems often use consensus (e.g., Paxos, Raft)
  - AP systems often use eventual consistency

- Many practical systems allow tuning:
  - MongoDB has configurable consistency
  - Cassandra has unable quorum reads/writes

Worksheet