Consistent Hashing Explained

Why traditional hashing doesn't work in distributed systems

ASHISH PRATAP SINGH

FEB 18, 2025

In a **distributed system** where nodes (servers) are frequently **added or removed**, efficiently routing requests becomes challenging.

A common approach is to use **hash the request** and assign it to a server using `Hash(key) mod N`, where N is the number of servers.

However, this method is highly dependent on the number of servers, and any change in N can lead to **significant rehashing**, causing a major redistribution of keys (requests).

Consistent hashing addresses this issue by ensuring that only a small subset of keys need to be reassigned when nodes are added or removed.

Popularized by [Amazon's Dynamo paper](), it has now become a fundamental technique

in distributed databases like DynamoDB, Cassandra and ScyllaDB.

In this article, we'll explore what consistent hashing is, why it's needed, how it works, and how to implement it in code.

If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

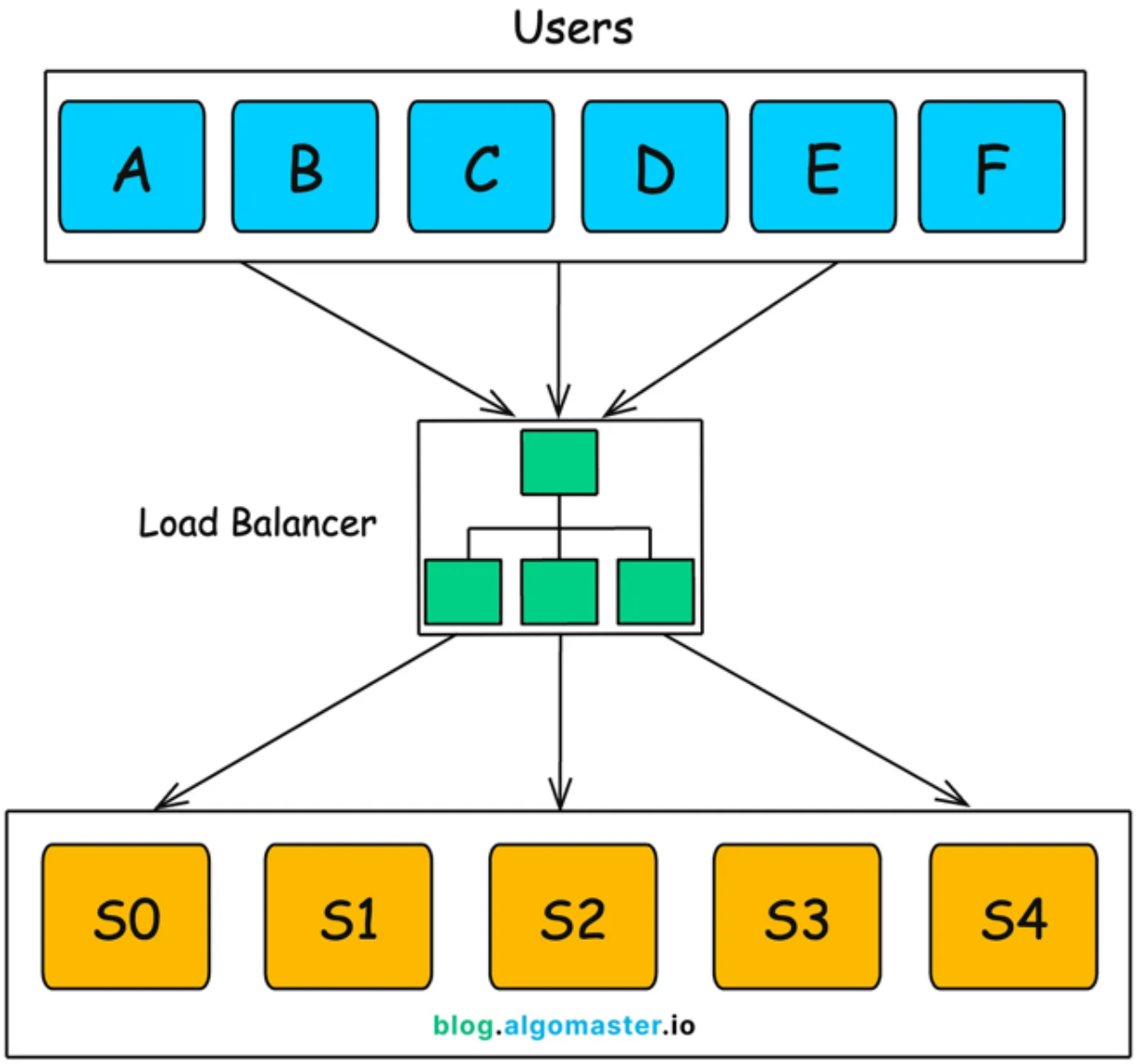
As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

---

1. The Problem with Traditional Hashing

Imagine you're building a **high-traffic web application** that serves millions of users daily. To handle the load efficiently, you distribute incoming requests across multiple backend servers using a **hash-based load balancer**.

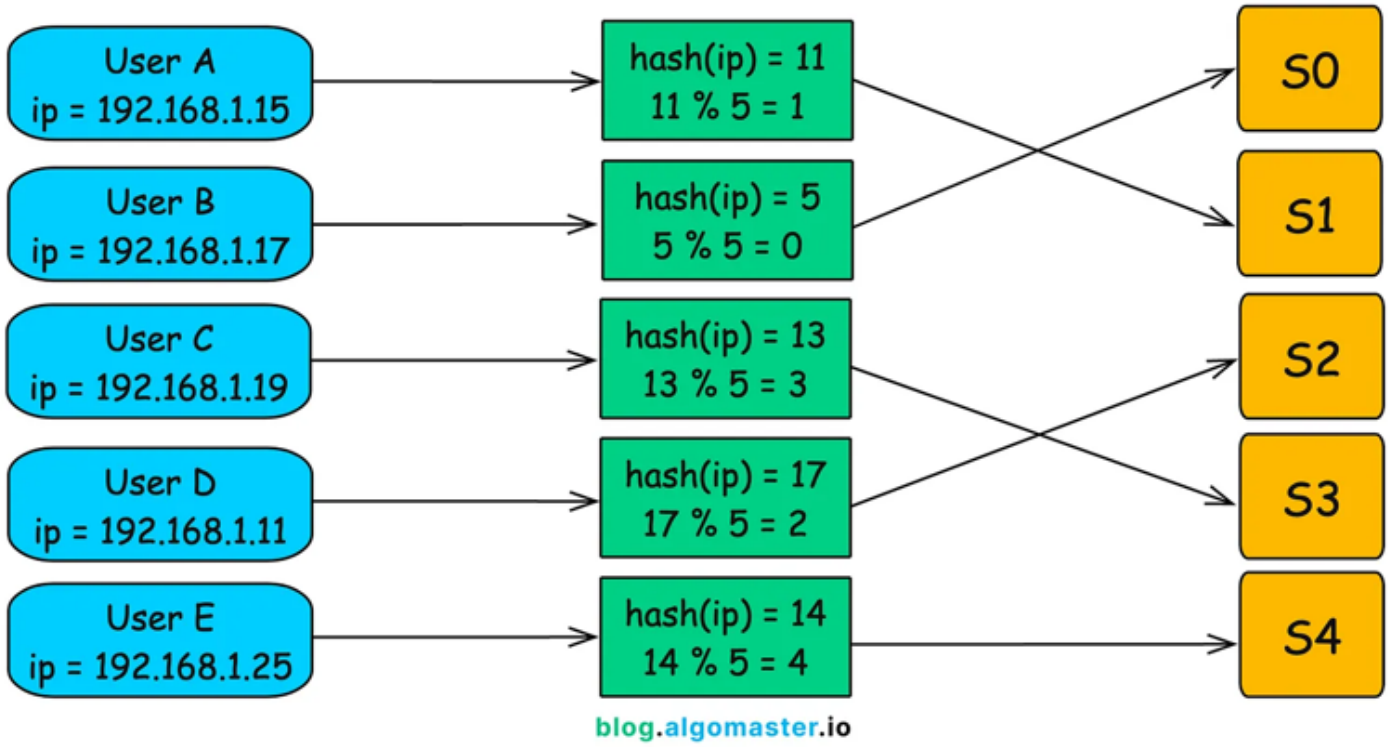
Your system consists of **5 backend servers** (`S0, S1, S2, S3, S4`), and requests are assigned using a hash function that maps each user's **IP address to a specific server**.



The process works like this:

1. The load balancer takes a user's IP address (or session ID).
2. A **hash function** maps the IP to one of the backend servers by taking the **sum of bytes in the IP address** and computing `mod 5` (since we have 5 servers).
3. The request is **routed to the assigned server**, ensuring that the same user is always directed to the same server for session consistency.

Example:

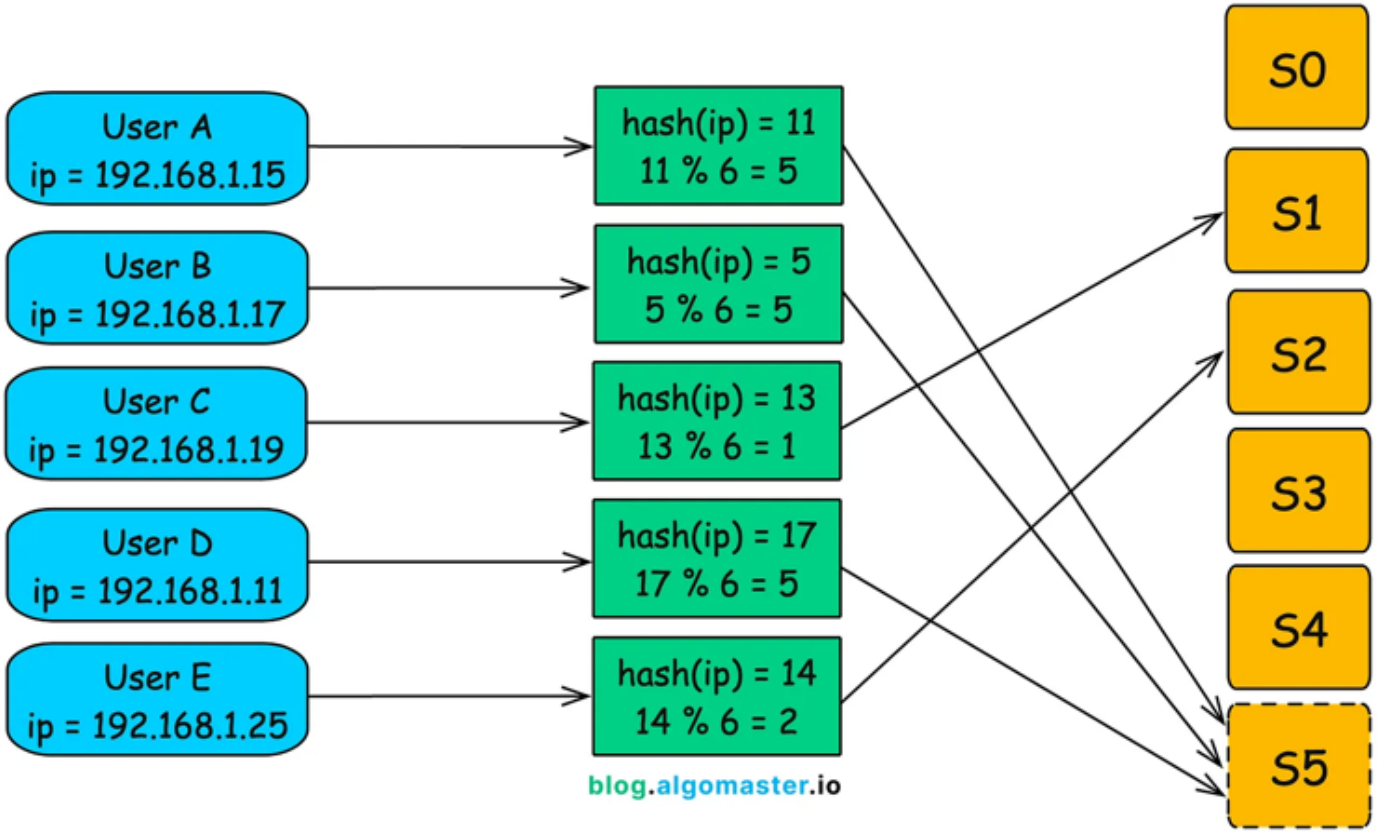


Everything Works Fine... Until You Scale

This approach works **as long as the number of servers remains constant**. But what happens when you **add or remove a server**?

Scenario 1: Adding a New Server (S5)

As traffic increases, you decide to **scale up** by adding a new backend server (S5). Now, the hash function must be modified to use `mod 6` instead of `mod 5` since we have 6 servers now.

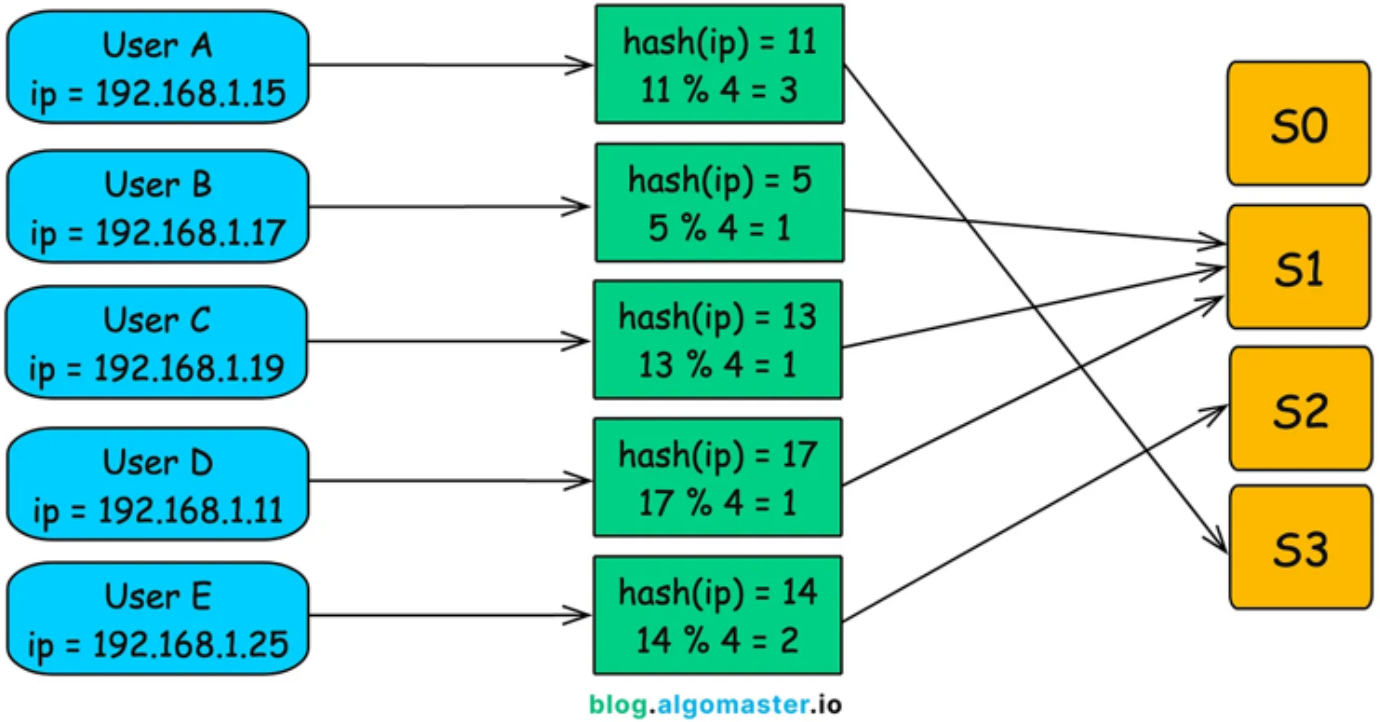


This seemingly simple change **completely disrupts the existing mapping**, causing **most users to be reassigned to different servers.**

This results into **massive rehashing**, leading to **high overhead, and potential downtime.**

Scenario 2: Removing a Server (S4)

Now, let's say one of the servers (S4) fails or is removed. The number of servers drops to 4, forcing the hash function to switch from `mod 5` to `mod 4`.



Even though only **one** server was removed, **most users are reassigned to different servers**. This can cause:

- **Session Loss:** Active users may be logged out or disconnected.
- **Cache invalidation:** Cached data becomes irrelevant, increasing database load.

- **Severe performance degradation:** The system may struggle to run efficiently.

The Solution: Consistent Hashing

Consistent hashing offers a more scalable and efficient solution by ensuring that only a **small fraction of users** are reassigned when scaling up or down.

It performs really well when operated in dynamic environments, where the system scales up and down frequently.

---

2. How Consistent Hashing Works

Consistent hashing is a **distributed hashing technique** used to efficiently distribute data across multiple nodes (servers, caches, etc.).

It uses a **circular hash space** (hash ring) with a large and constant hash space.

Both nodes (servers, caches, or databases) and keys (data items) are mapped to positions on this hash ring using a **hash function**.

Unlike modulo-based hashing, where changes in the number of nodes cause large-

scale remapping, consistent hashing ensures that only a small fraction of keys are reassigned when a node is added or removed, making it highly scalable and efficient.

> In consistent hashing, when the number of nodes changes, only `k/n` keys need to be reassigned, where `k` is the total number of keys and `n` is the total number of nodes.

2.1 Constructing the Hash Ring

Instead of distributing keys based on `Hash(key) mod N`, consistent hashing places both servers and keys on a circular hash ring.

Defining the Hash Space

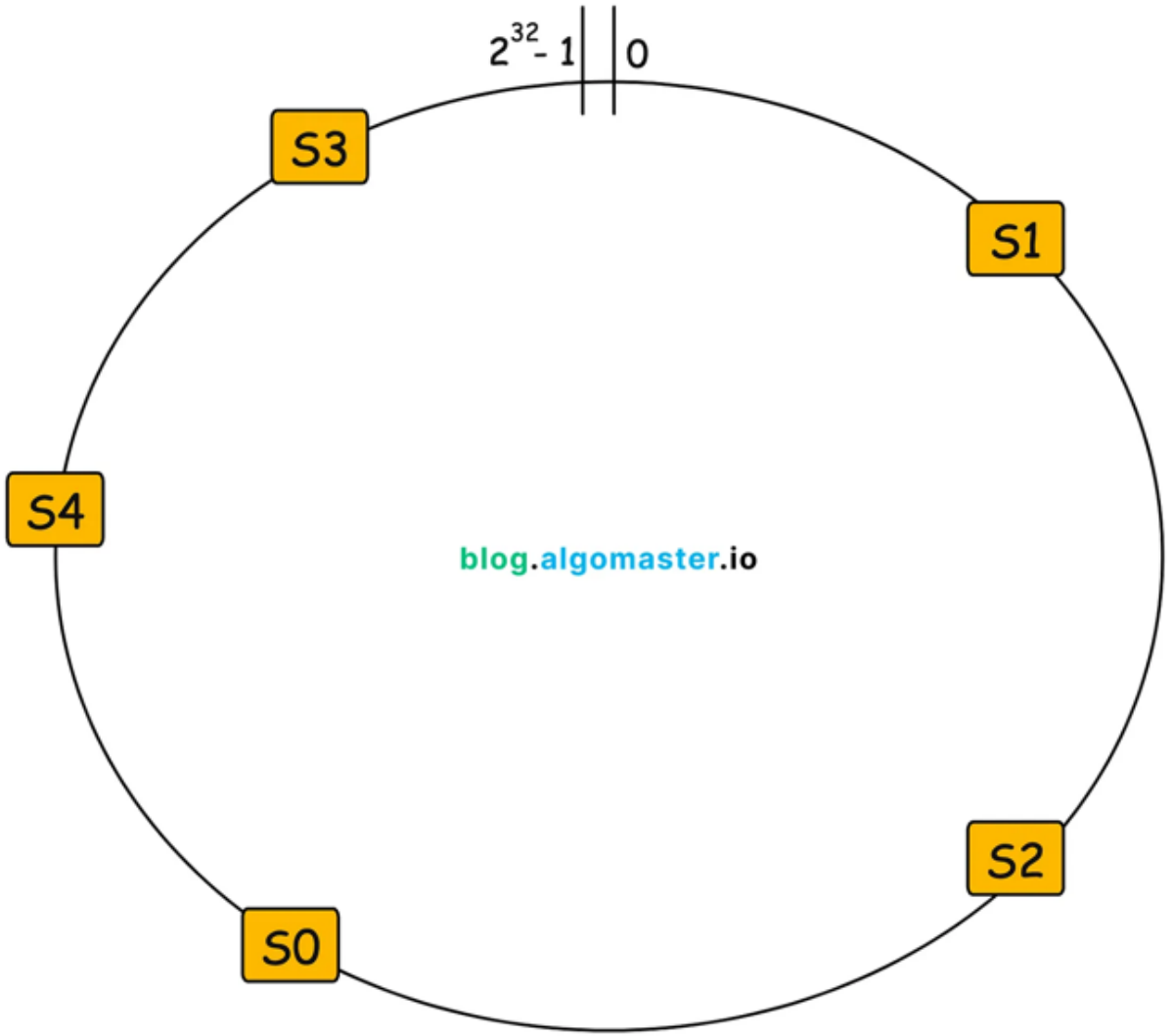
- We use a large, fixed hash space ranging from `0` to `2^32 - 1` (assuming a 32-bit hash function).
- This creates a circular structure, where values wrap around after reaching the maximum limit.

Placing Servers on the Ring

- Each **server** (**node**) is assigned a position on the hash ring by computing

`Hash(server_id)`.

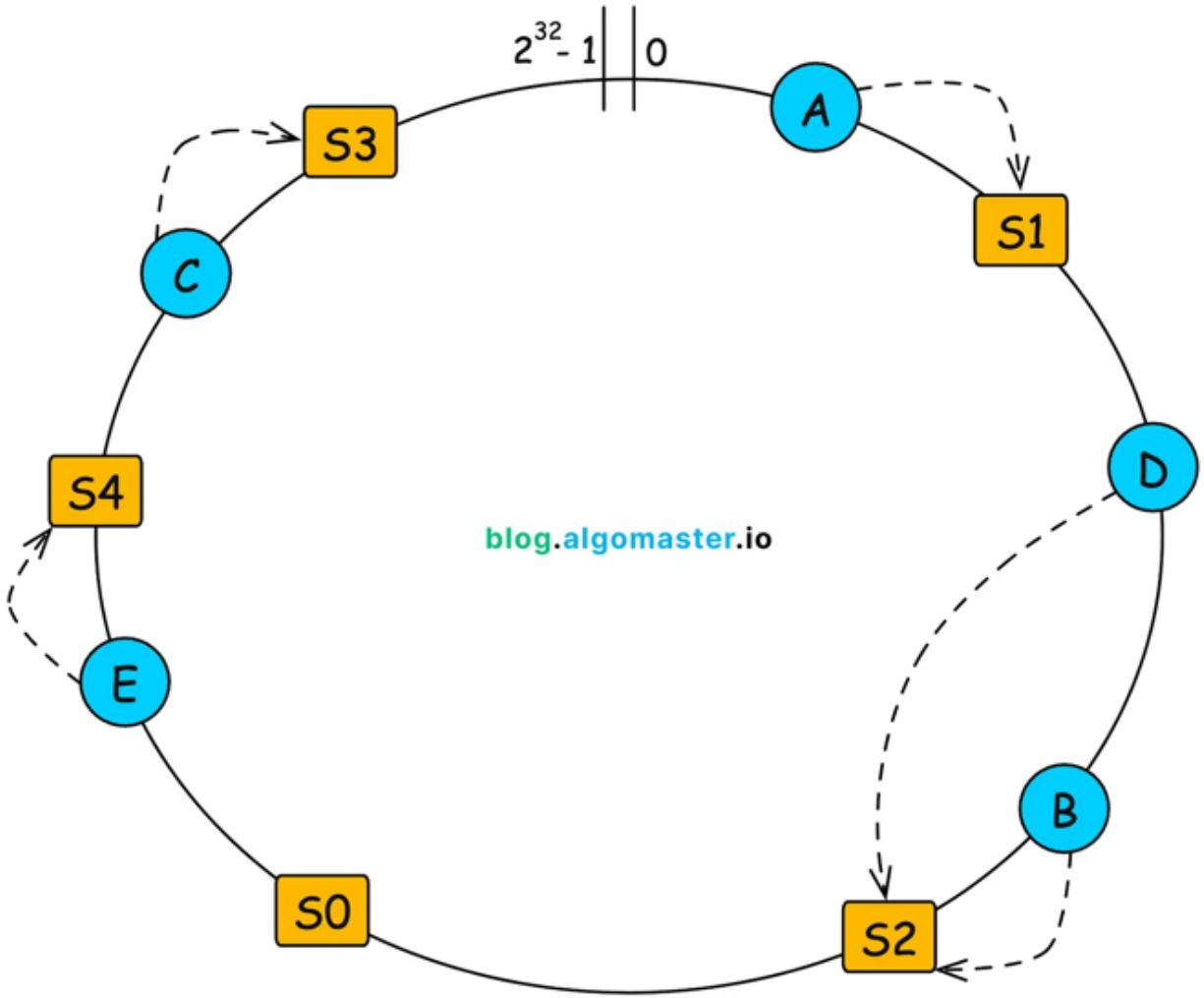
- Using the above example with **5 servers** (`S0, S1, S2, S3, S4`), the hash function distributes them at different positions around the ring.



Mapping Keys to Servers

- When a key is added, its position is determined by computing `Hash(key)`.

    - Example: a user's request is assigned a position on the ring based on the hash of their IP address: `Hash(IP Address)`
- We then move clockwise around the ring until we find the next available server.
- The key (or request) is assigned to this server for storage or retrieval.

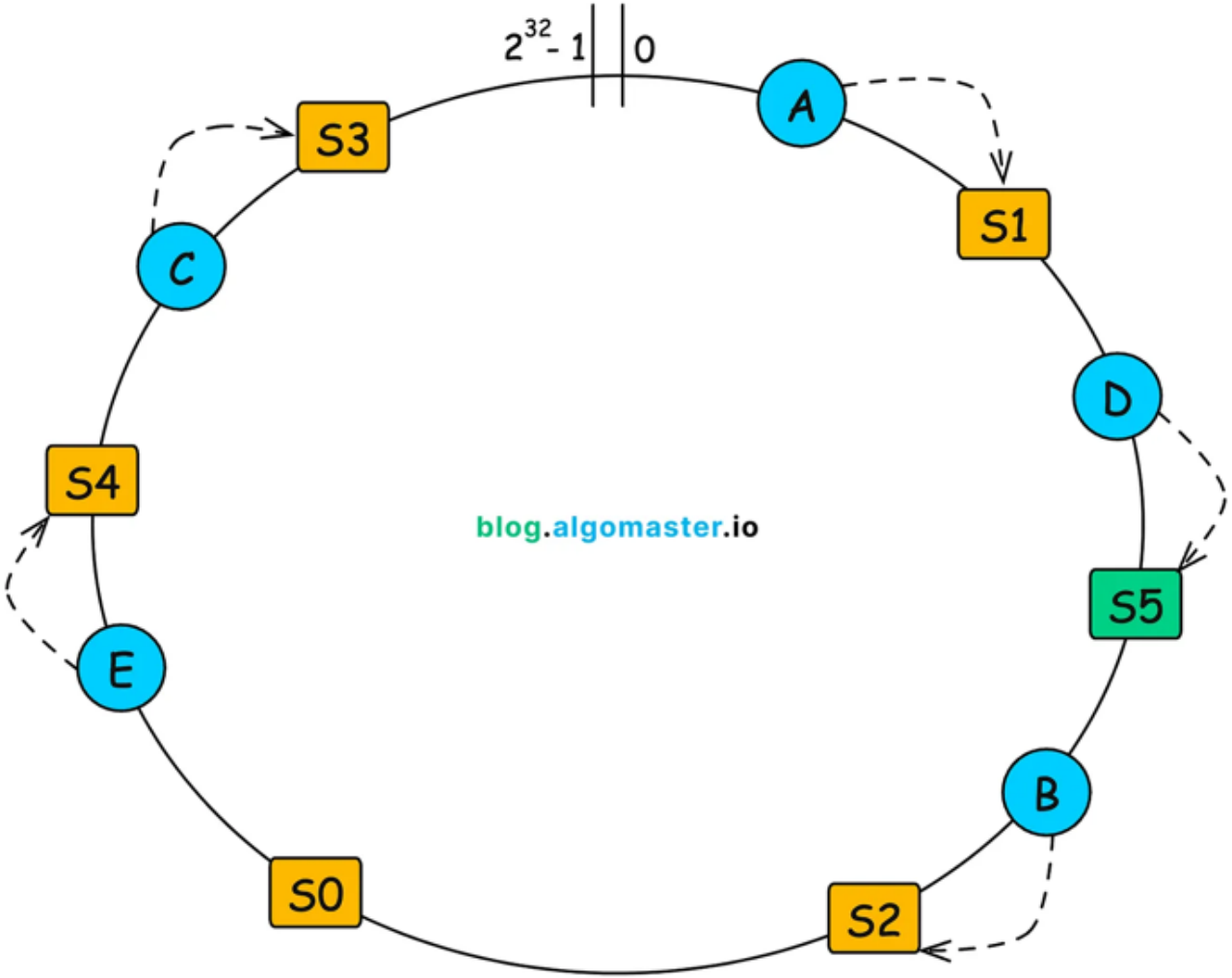


> **Note:** In case a key's hash falls directly on a node's position, it belongs to that node.

2.2 Adding a New Server

Suppose we add a **new server** (`S5`) to the system.

- The position of `S5` falls between `S1` and `S2` in the hash ring.
- `S5` takes over all keys (requests) that fall between `S1` and `S5`, which were previously handled by `S2`.
    - **Example:** `User D's` requests which were originally assigned to `S2,` will now be redirected to `S5.`

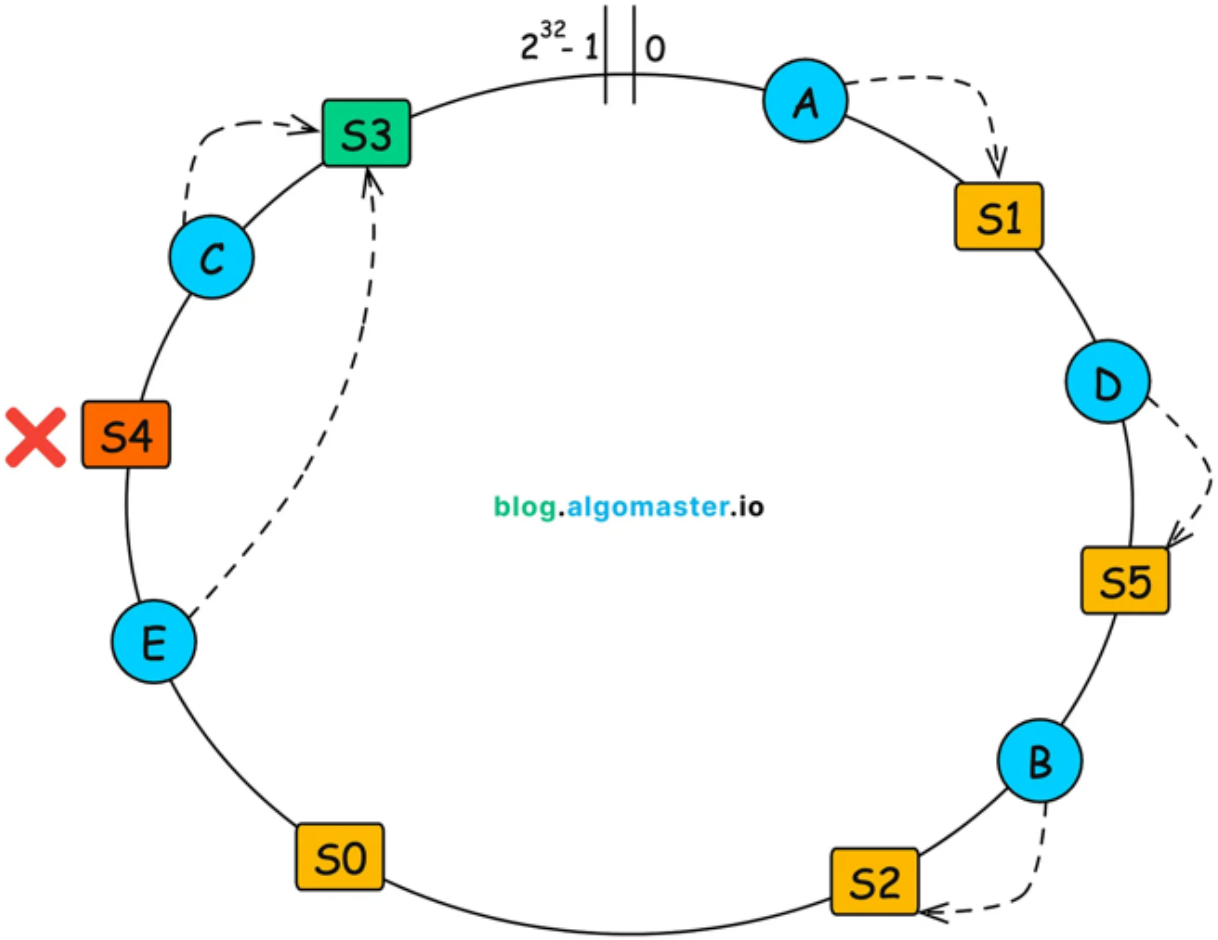


This demonstrates how consistent hashing efficiently redistributes keys with minimal disruption, ensuring that only a small subset of keys are reassigned when new servers are added.

2.3 Removing a Node

When a server, such as S4, fails or is removed from the system:

- All keys previously assigned to S4 are reassigned to the next available server in the ring (S3).
- Only the keys (requests) that were mapped to S4 need to move, while all other keys remain unaffected.



This results in **minimal data movement**, unlike traditional hashing where removing a node would require reassigning most keys.

3. Virtual Nodes

In **basic consistent hashing**, each server is assigned **a single position** on the hash ring. However, this can lead to **uneven data distribution**, especially when:

- The number of servers is small.
- Some servers accidentally get clustered together, creating **hot spots**.
- A server is **removed**, causing a sudden load shift to its immediate neighbor.

Virtual nodes (**VNodes**) are a technique used in consistent hashing to improve **load balancing** and **fault tolerance** by distributing data more evenly across servers.

How Virtual Nodes Work

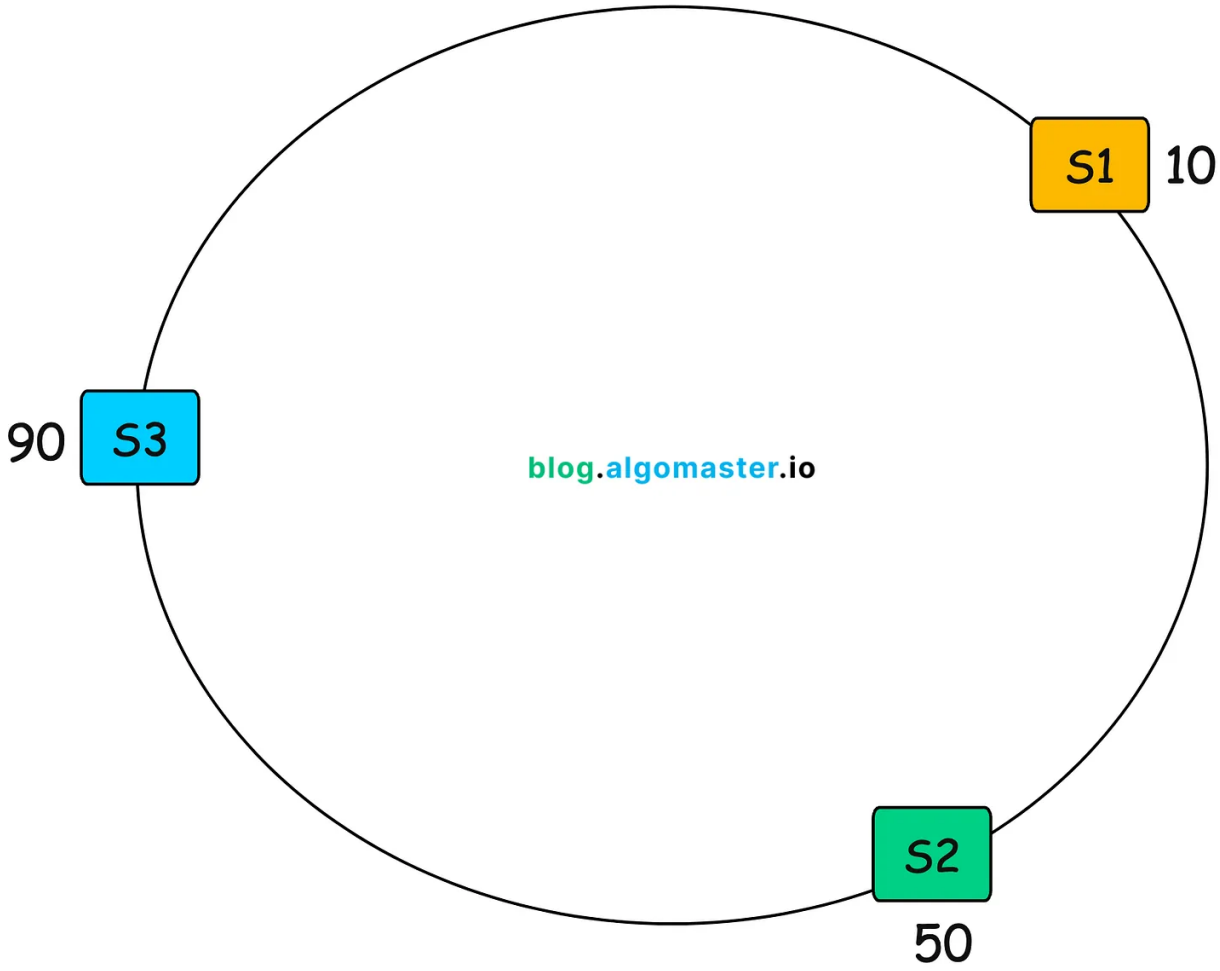
Instead of assigning one position per server, each physical server is assigned **multiple positions** (virtual nodes) on the hash ring.

- Each server is hashed multiple times to different locations on the ring.
- When a request (key) is hashed, it is assigned to the next virtual node in a clockwise direction.
- The request is then routed to the actual server associated with the virtual node.

Example

Assume we have **three physical servers** (**S1, S2, S3**). Without virtual nodes, their positions might be:

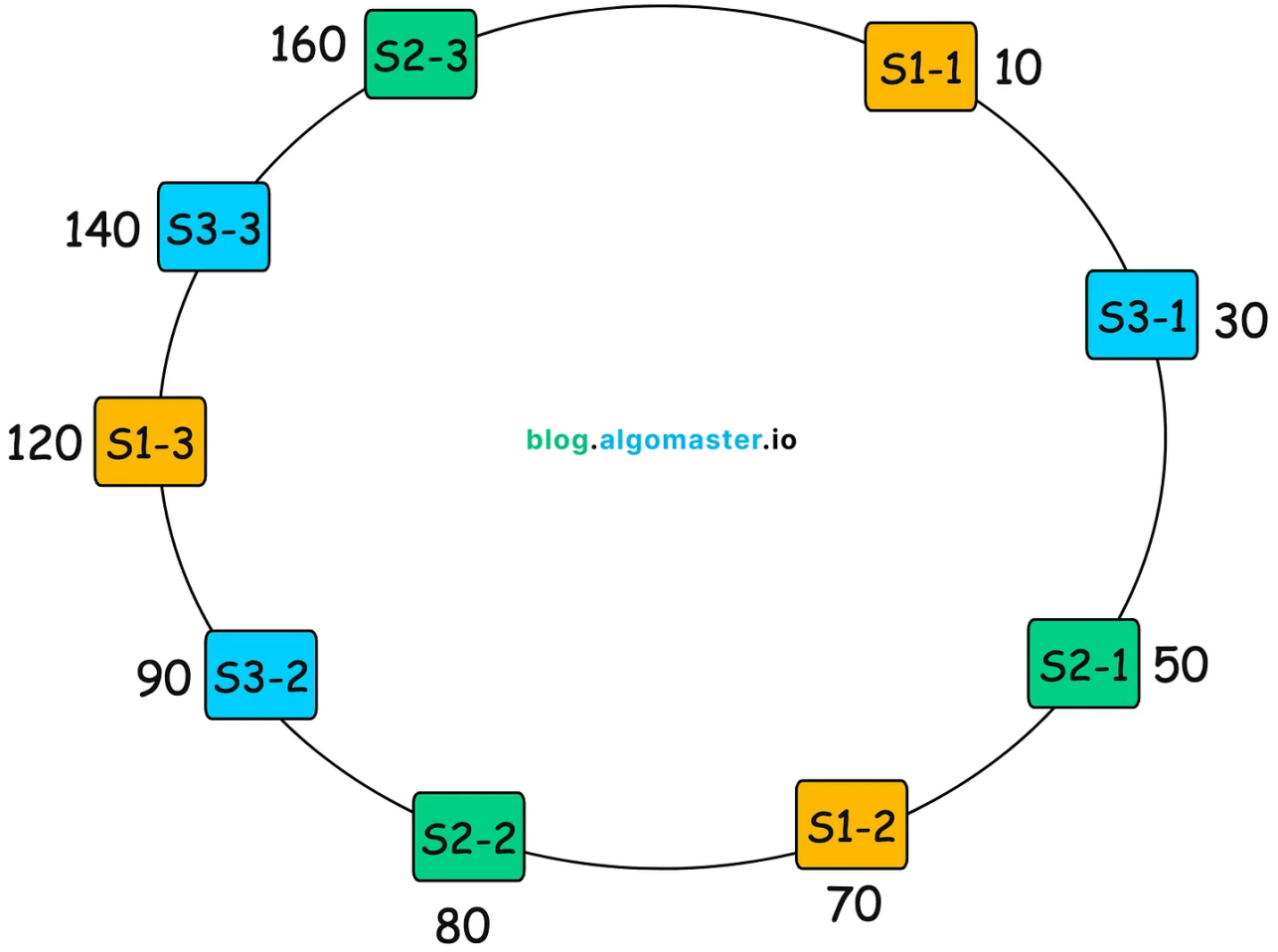
```
S1 → Position 10
S2 → Position 50
S3 → Position 90
```



If **S1 fails**, all its keys must be reassigned to **S2**, which can create an overload.

With **virtual nodes**, each server is hashed multiple times:

```
S1–1 → Position 10
S1–2 → Position 70
S1–3 → Position 120
S2–1 → Position 50
S2–2 → Position 80
S2–3 → Position 160
S3–1 → Position 30
S3–2 → Position 90
S3–3 → Position 140
```



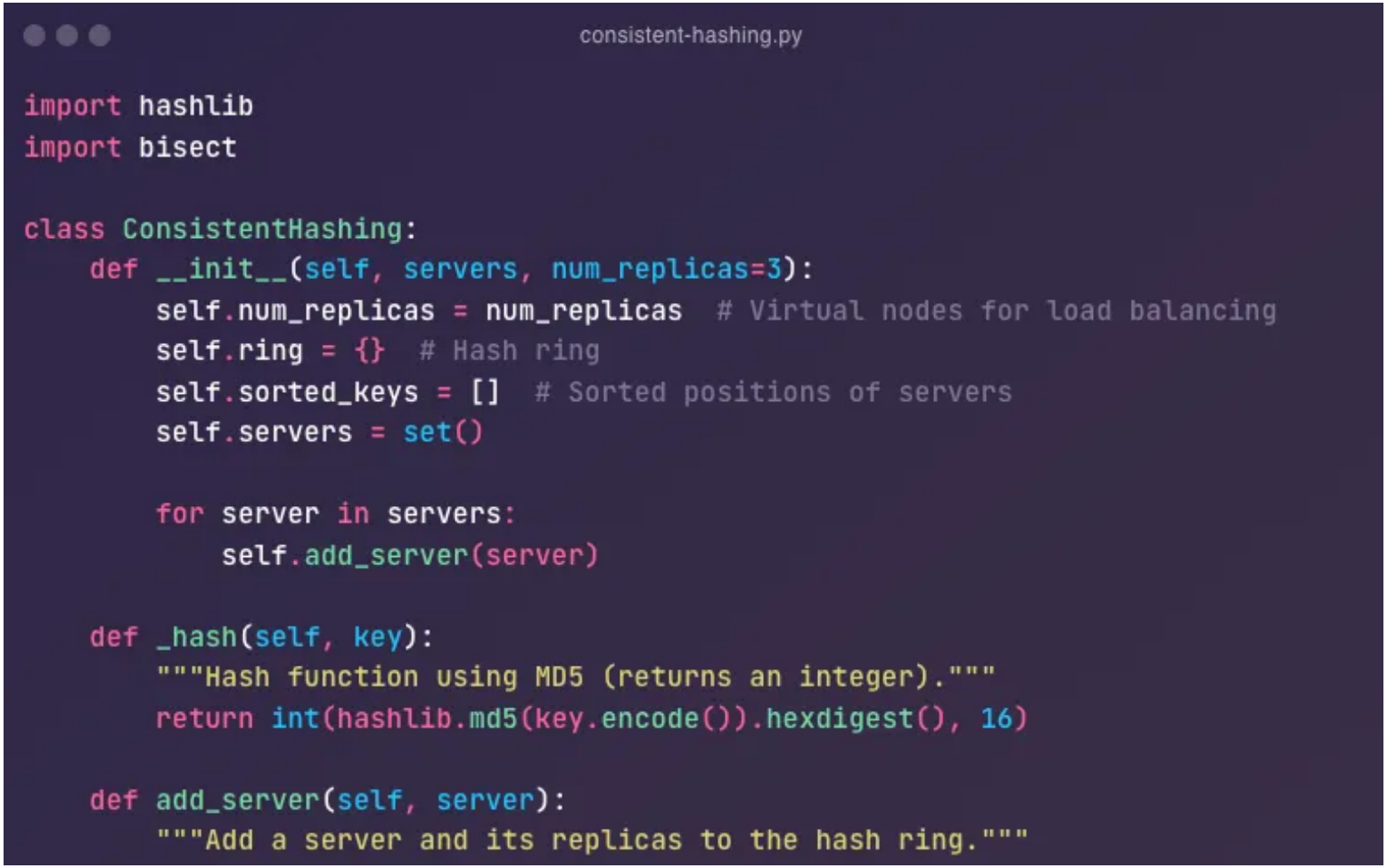
Now, instead of just one point, S1 is represented at **multiple positions**, making the distribution **more even**.

If **S1 fails**, its keys are more evenly redistributed among **S2 and S3**, rather than all

going to **S2**.

---

4. Code Implementation (Python)



```
        self.servers.add(server)
        for i in range(self.num_replicas):
            hash_val = self._hash(f"{server}-{i}")
            self.ring[hash_val] = server
            bisect.insort(self.sorted_keys, hash_val)

    def remove_server(self, server):
        """Remove a server and its replicas from the hash ring."""
        if server in self.servers:
            self.servers.remove(server)
            for i in range(self.num_replicas):
                hash_val = self._hash(f"{server}-{i}")
                self.ring.pop(hash_val, None)
                self.sorted_keys.remove(hash_val)

    def get_server(self, key):
        """Find the closest server for the given key."""
        if not self.ring:
            return None
        hash_val = self._hash(key)
        index = bisect.bisect(self.sorted_keys, hash_val) % len(self.sorted_keys)
        return self.ring[self.sorted_keys[index]]

# Initialize with servers
servers = ["S0", "S1", "S2", "S3", "S4", "S5"]
ch = ConsistentHashing(servers)
```

Code Links: Python, Java

Explanation:

1. **Key Components**
    - **Hash Ring** (`self.ring`): Stores hash values → server mappings. Uses **virtual nodes** (replicas) for better load balancing.
    - **Sorted Keys** (`self.sorted_keys`): Maintains a **sorted list** of hash values for efficient lookups.
    - **Server Set** (`self.servers`): Tracks active physical servers.
2. **Initialization** (`__init__`)
    - Calls `add_server()` for each server, hashing it multiple times (based on `num_replicas`) to ensure even distribution.
3. **Hashing Function** (`_hash`)
    - Uses **MD5 hashing** to convert strings into large integers for consistent placement on the hash ring.
4. **Adding a Server** (`add_server`)
    - Generates multiple hash values for each server (`server-0`, `server-1`, etc.).

- Stores these in `self.ring` and maintains a sorted order in `self.sorted_keys` for fast lookup.

5. **Removing a Server** (`remove_server`)
    - Deletes the server's hash values and its virtual nodes from `self.ring` and `self.sorted_keys`.
6. **Getting a Server** (`get_server`)
    - Hashes the input key and finds the closest **clockwise** server using `bisect.bisect()`. Wraps around to the first node if necessary.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn and X to stay updated.

Checkout my GitHub repositories for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish

Previous

Next

Discussion about this post

Comments Restacks

Write a comment...

Hervé BAKONGO 7 Oct 2024

Liked by Ashish Pratap Singh

Thanks, It is the Best article I have read about CAP THEOREM.

LIKE (2) REPLY SHARE

Ankit Singh 3 May

Liked by Ashish Pratap Singh

This is great, one of the best articles on CAP I have read so far.

LIKE (1) REPLY SHARE

15 more comments...