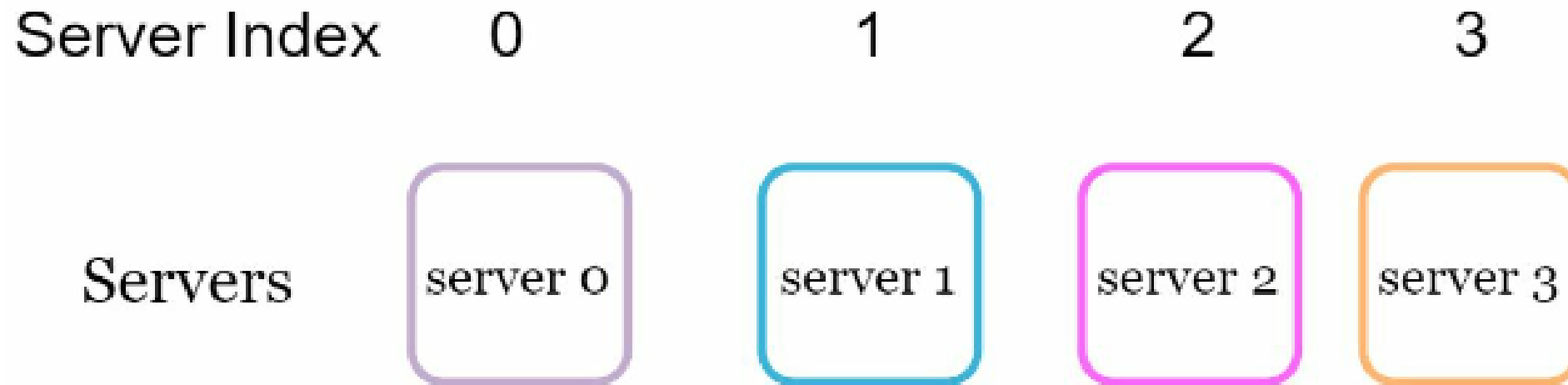


# Consistent Hashing

# Overview

- Rehashing Problem (modular hash)
- Consistent Hashing

## Cache Servers




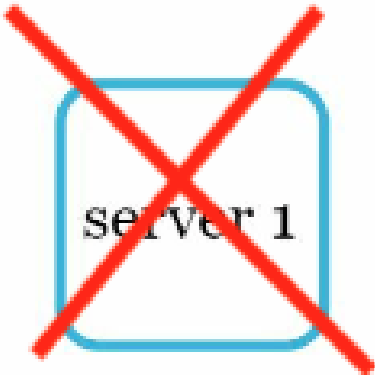


key		hash	hash % 4
key0	$f(\text{key0}) =$	18358617	1
key1	$f(\text{key1}) =$	26143584	0
key2		18131146	2
key3		35863496	0
key4		34085809	1
key5		27581703	3
key6		38164978	2
key7		22530351	3

$$\text{serverIndex} = \text{hash} \% 4$$

Server Index	0	1	2	3
Servers	server 0	server 1	server 2	server 3
Keys	key1 key3	key0 key4	key2 key6	key5 key7





## Server1 が Offline になったとき

$$\text{serverIndex} = \text{hash} \% 4$$

Server Index	0	1	2	3
Servers	 server 0	 server 1	 server 2	 server 3
Keys	key1 key3	key0 key4	key2 key6	key5 key7

key	Hash	hash % 3
key0	18358617	0
key1	26143584	0
key2	18131146	1
key3	35863496	2
key4	34085809	1
key5	27581703	0
key6	38164978	1
key7	22530351	0

$$\text{serverIndex} = \text{hash} \% 3$$

Server Index	0		1	2
Servers				
Keys	<b>key0</b> key1 <b>key5</b> <b>key7</b>		key2 <b>key4</b> key6	<b>key3</b>

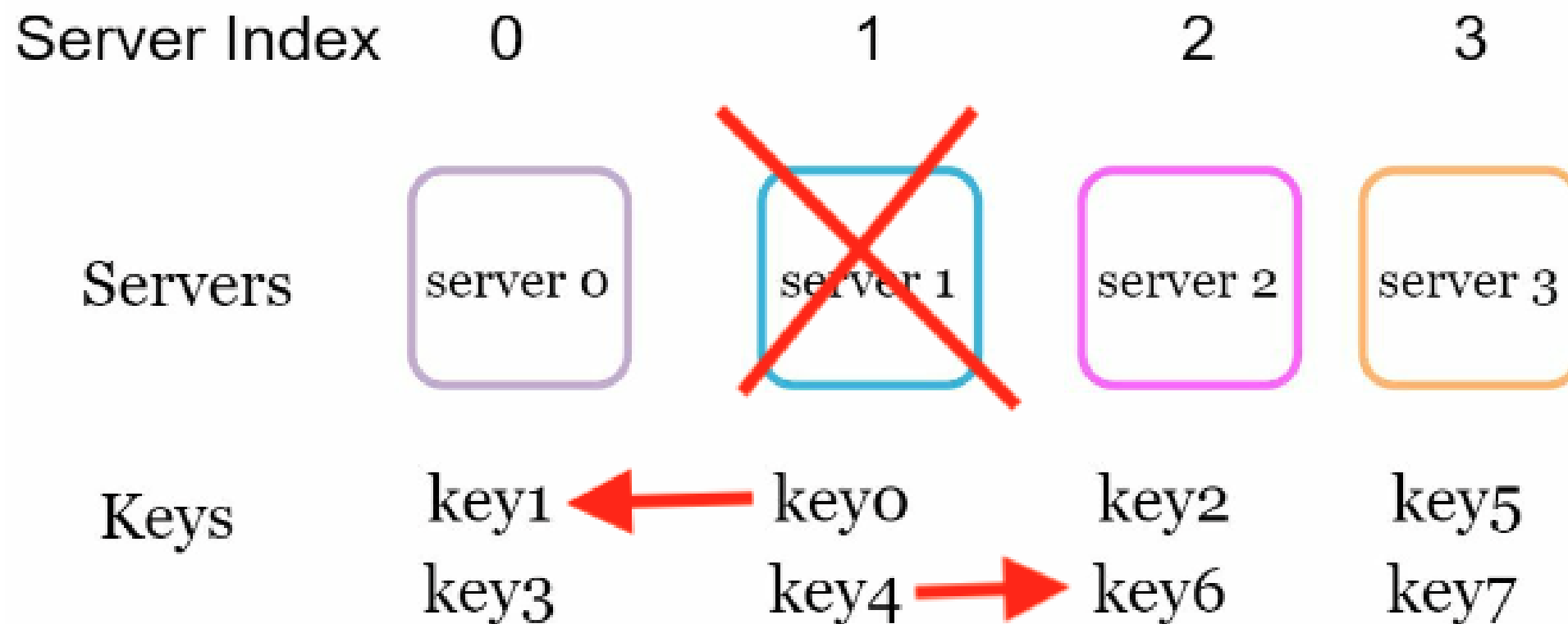


# Problem

データ量が多いと再配置に時間がかかる  
データの偏りが生じる

# 理想

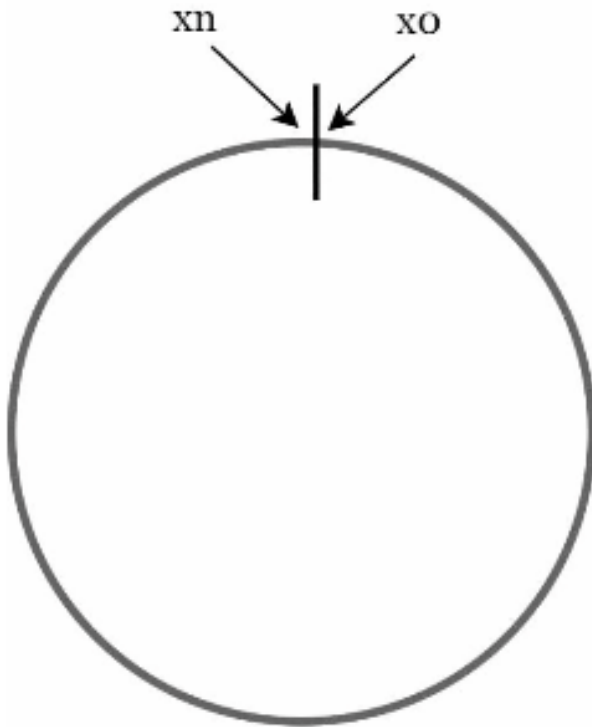
$$\text{serverIndex} = \text{hash} \% 4$$



# 実装

f: hash 関数

$x_0, x_1 \dots x_n$ : 出力範囲



- <https://github.com/Cyan4973/xxHash> など軽量なものがオススメ

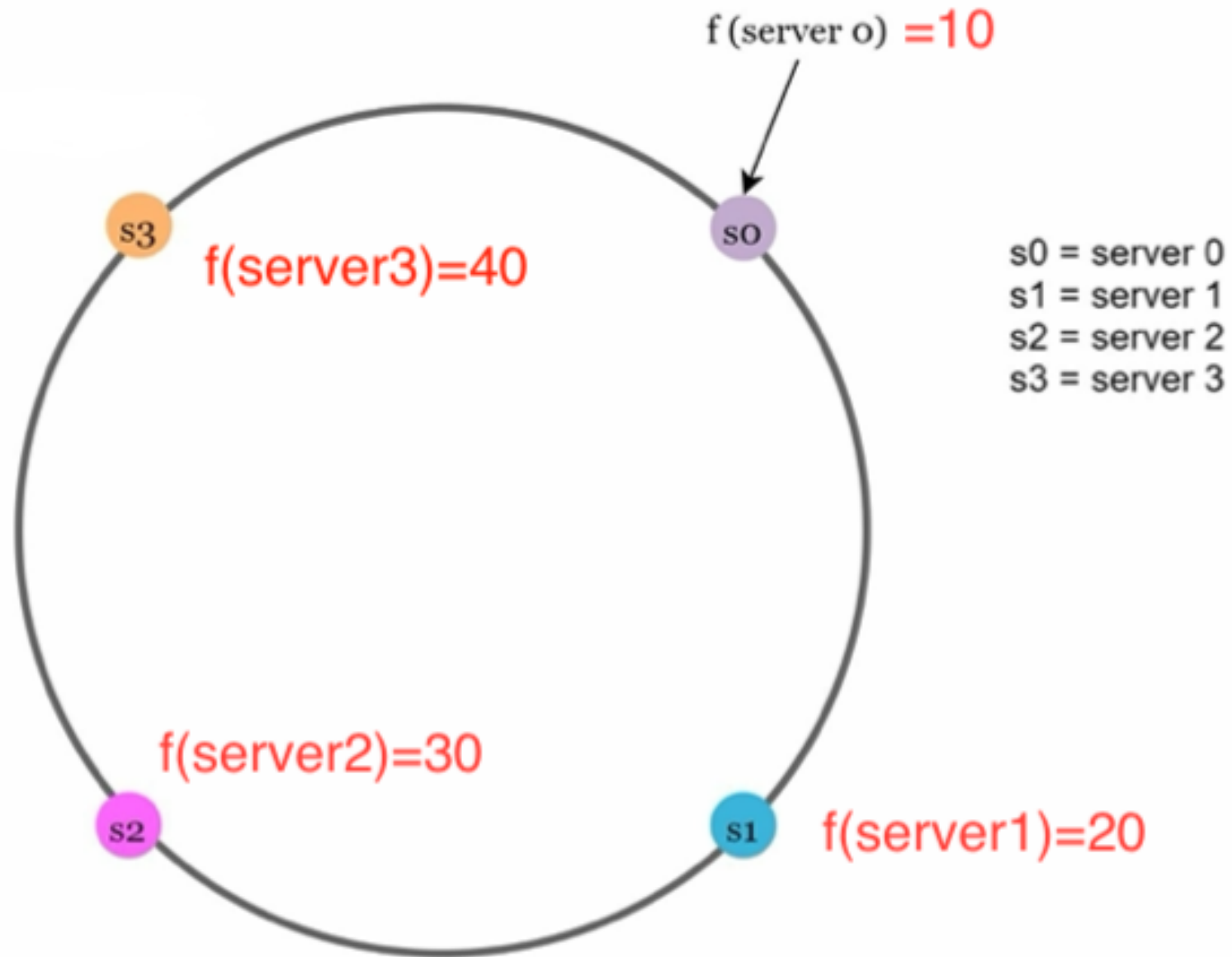
### Servers

server 0

server 1

server 2

server 3



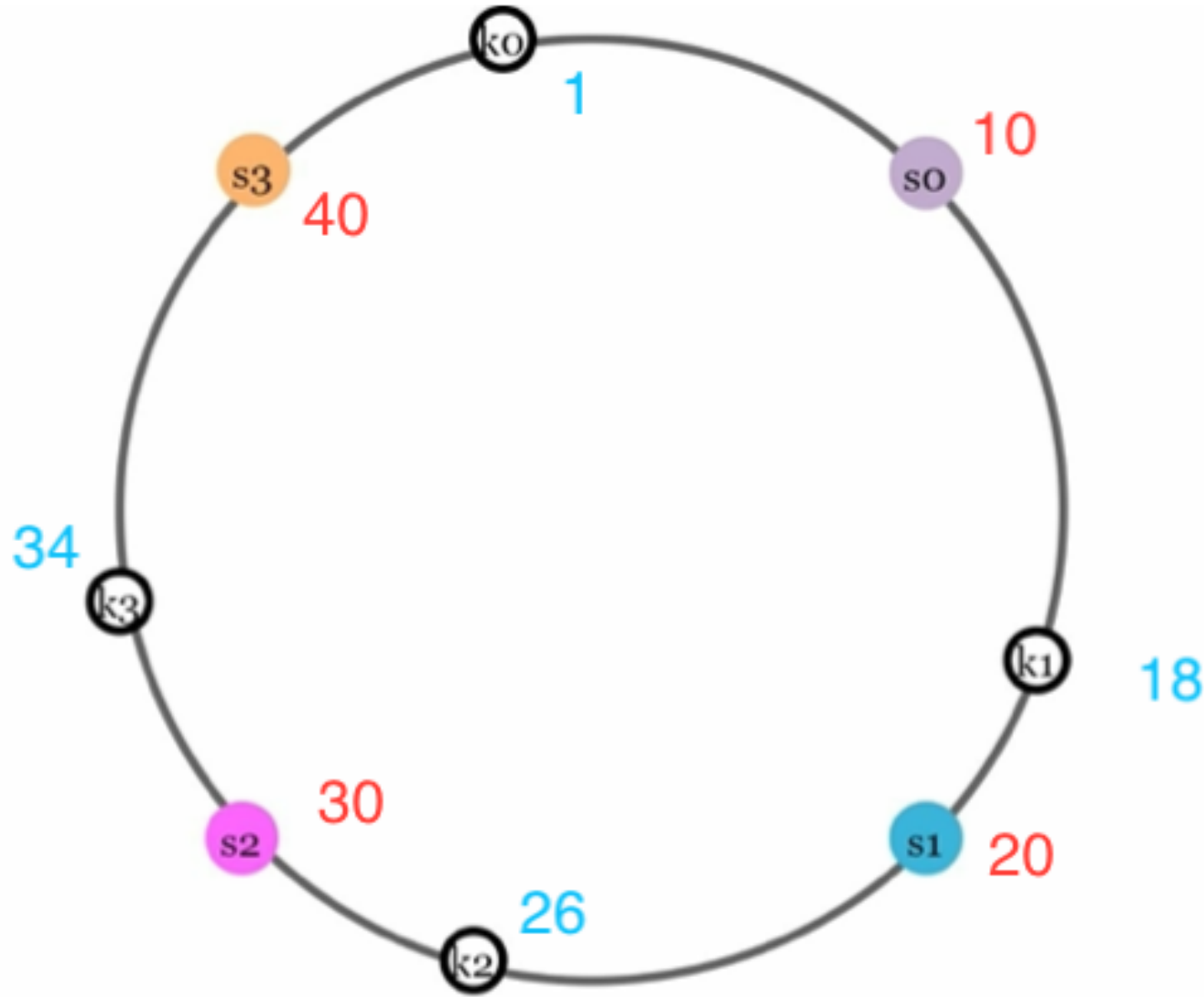
### Servers

server 0

server 1

server 2

server 3



s0 = server 0  
s1 = server 1  
s2 = server 2  
s3 = server 3

k0 = key0  
k1 = key1  
k2 = key2  
k3 = key3

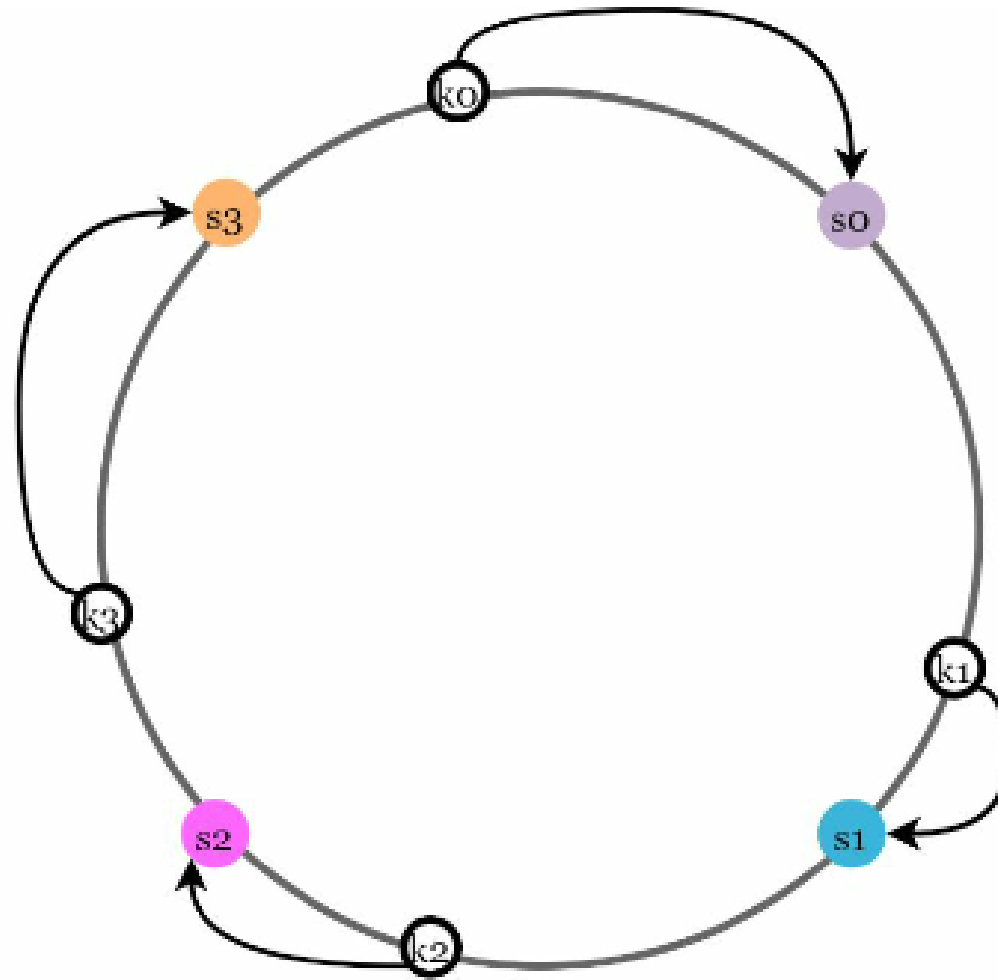
Servers

server 0

server 1

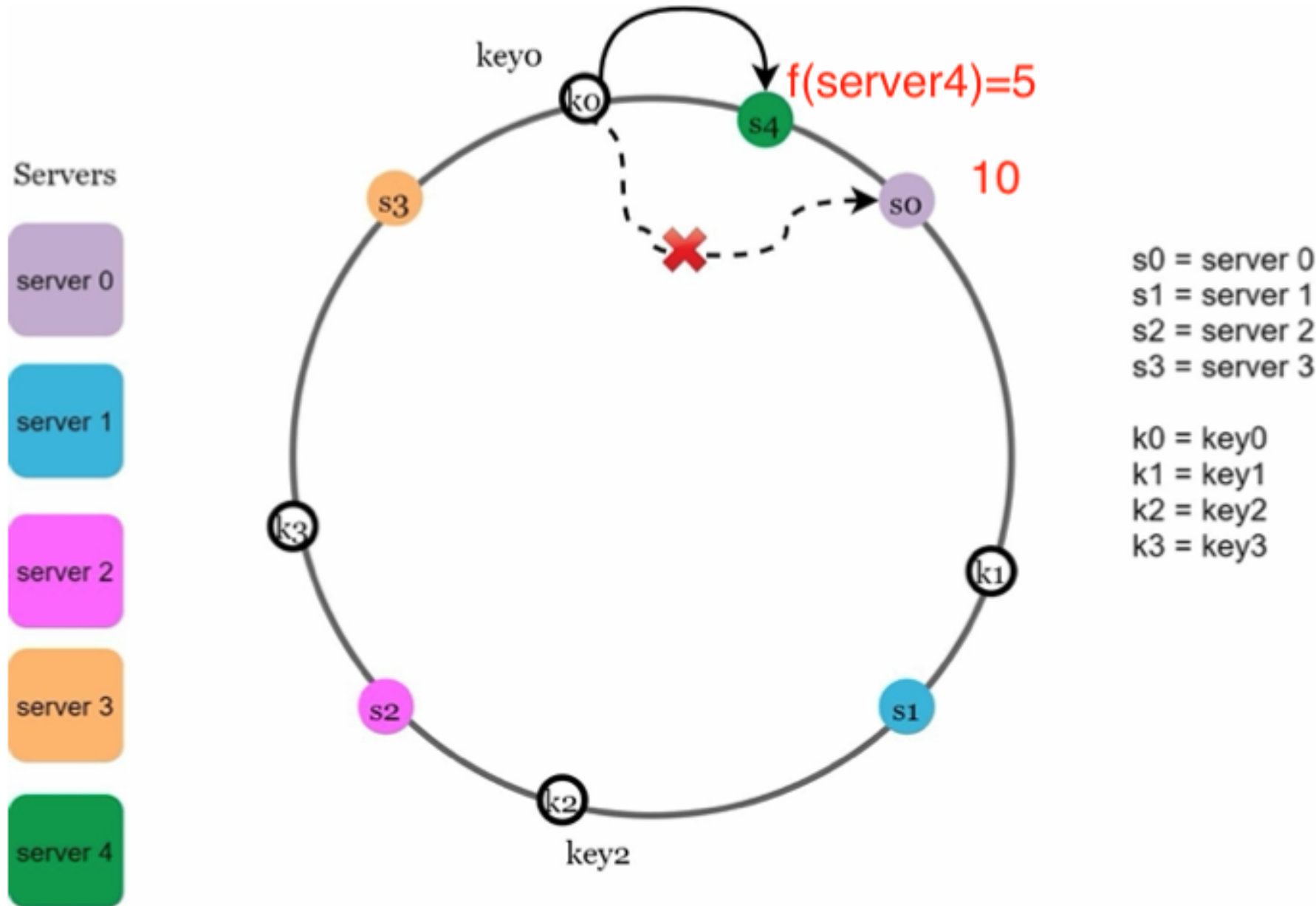
server 2

server 3

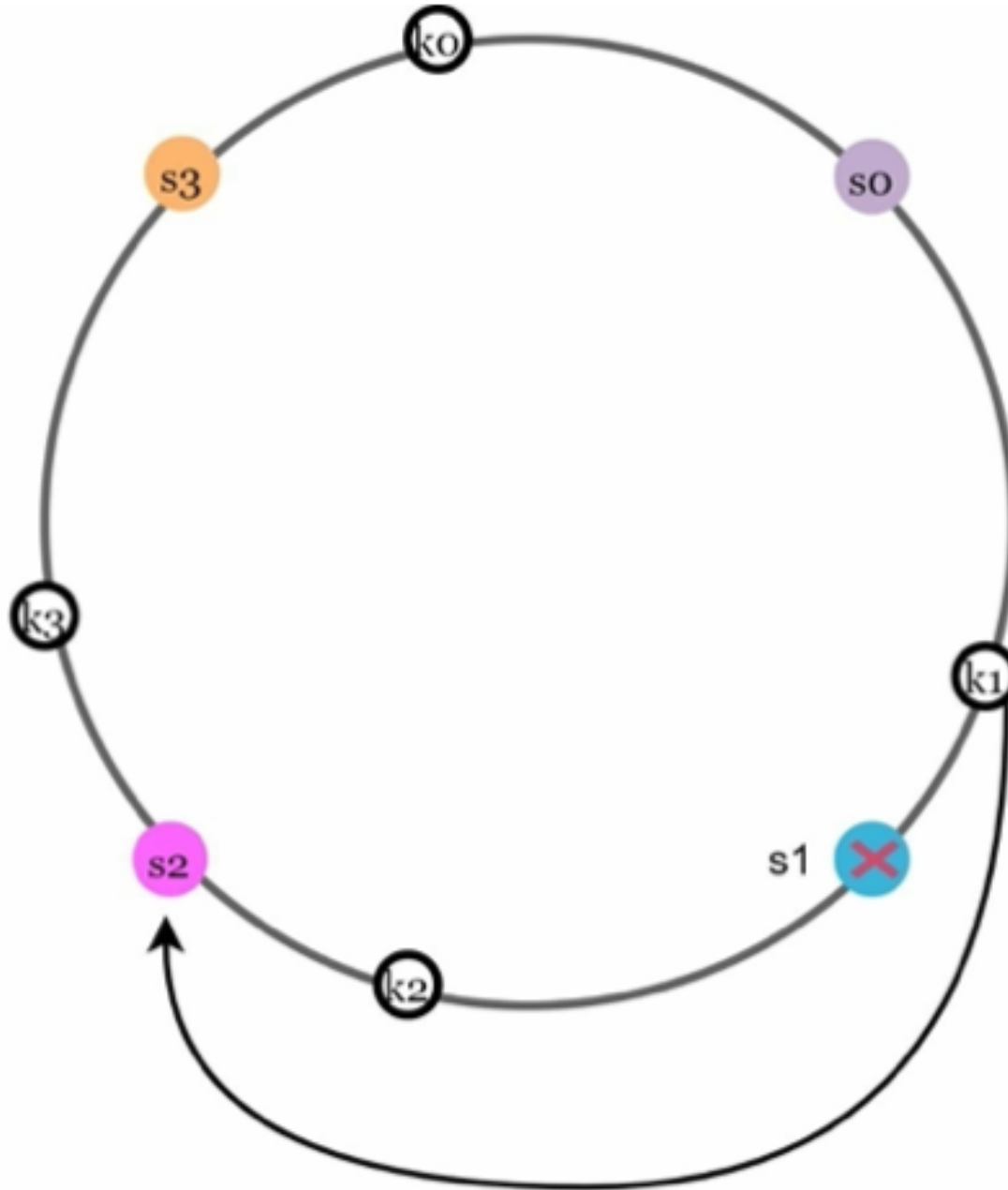


s0 = server 0  
s1 = server 1  
s2 = server 2  
s3 = server 3

k0 = key0  
k1 = key1  
k2 = key2  
k3 = key3



## Servers

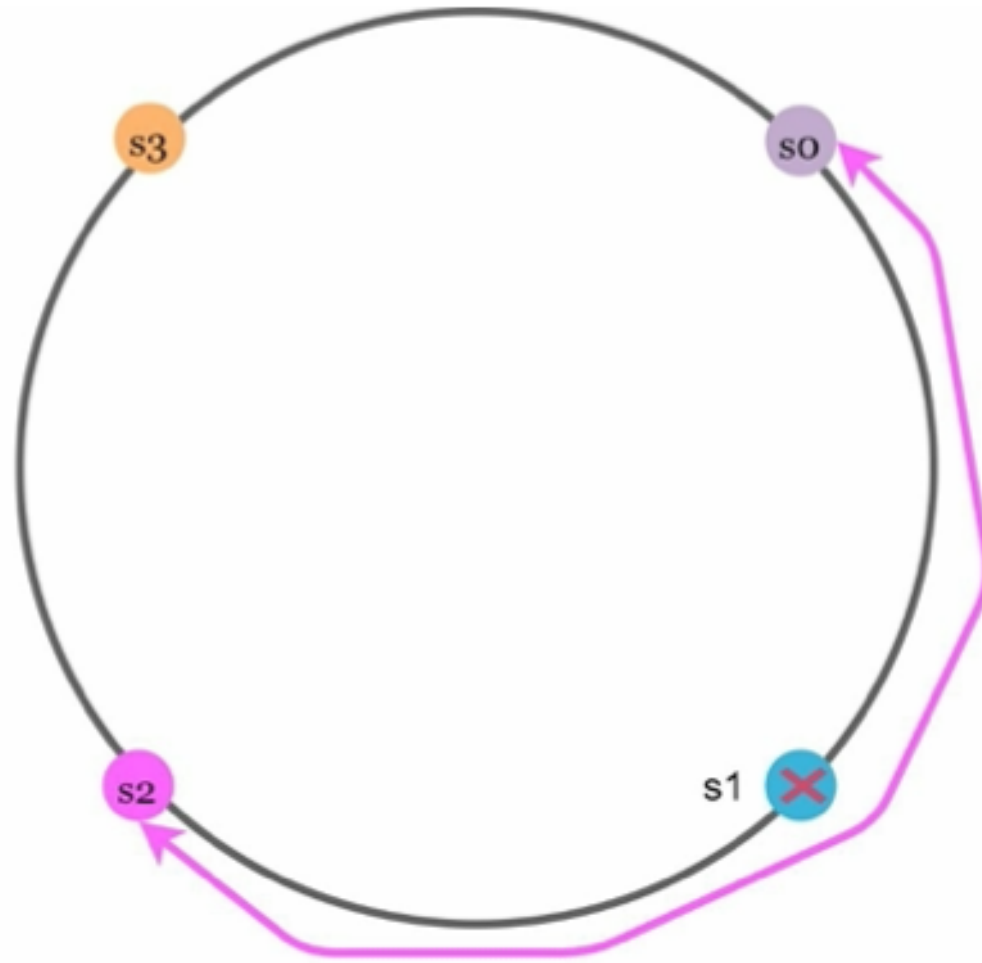
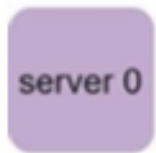


s0 = server 0  
s1 = server 1  
s2 = server 2  
s3 = server 3

k0 = key0  
k1 = key1  
k2 = key2  
k3 = key3

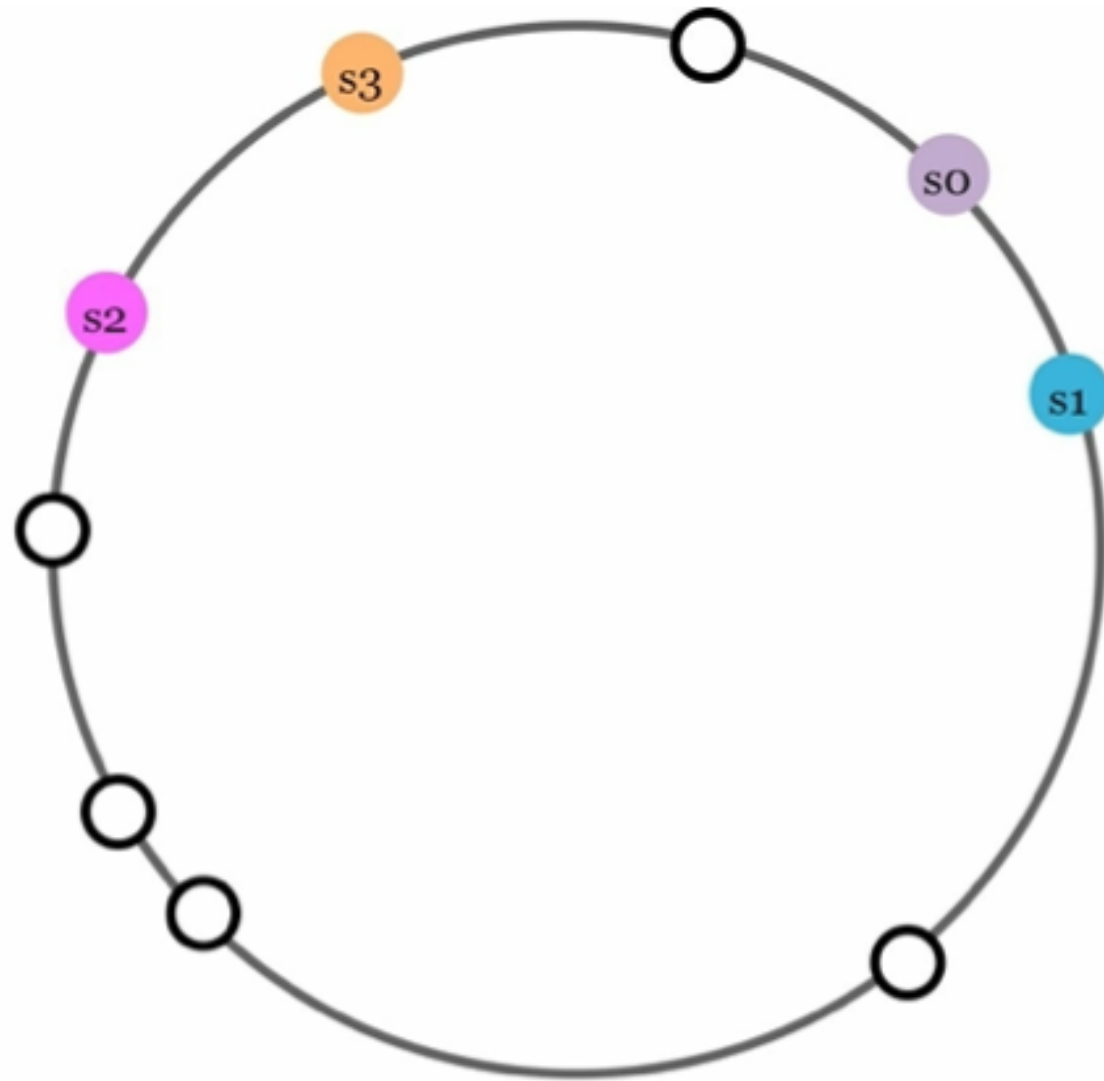
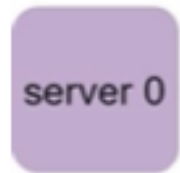


### Servers



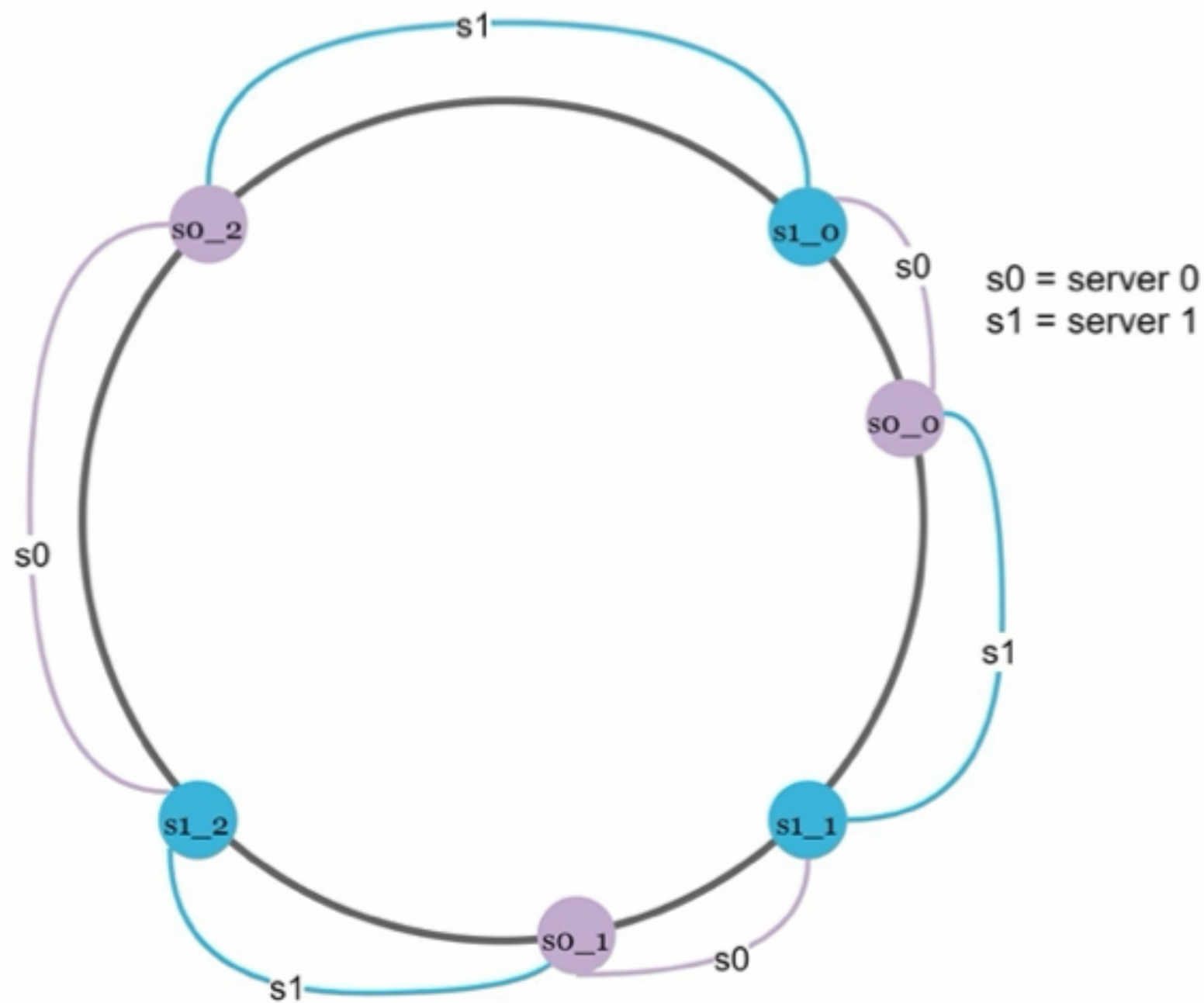
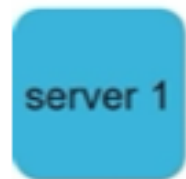
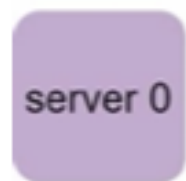
s0 = server 0  
s1 = server 1  
s2 = server 2  
s3 = server 3

## Servers

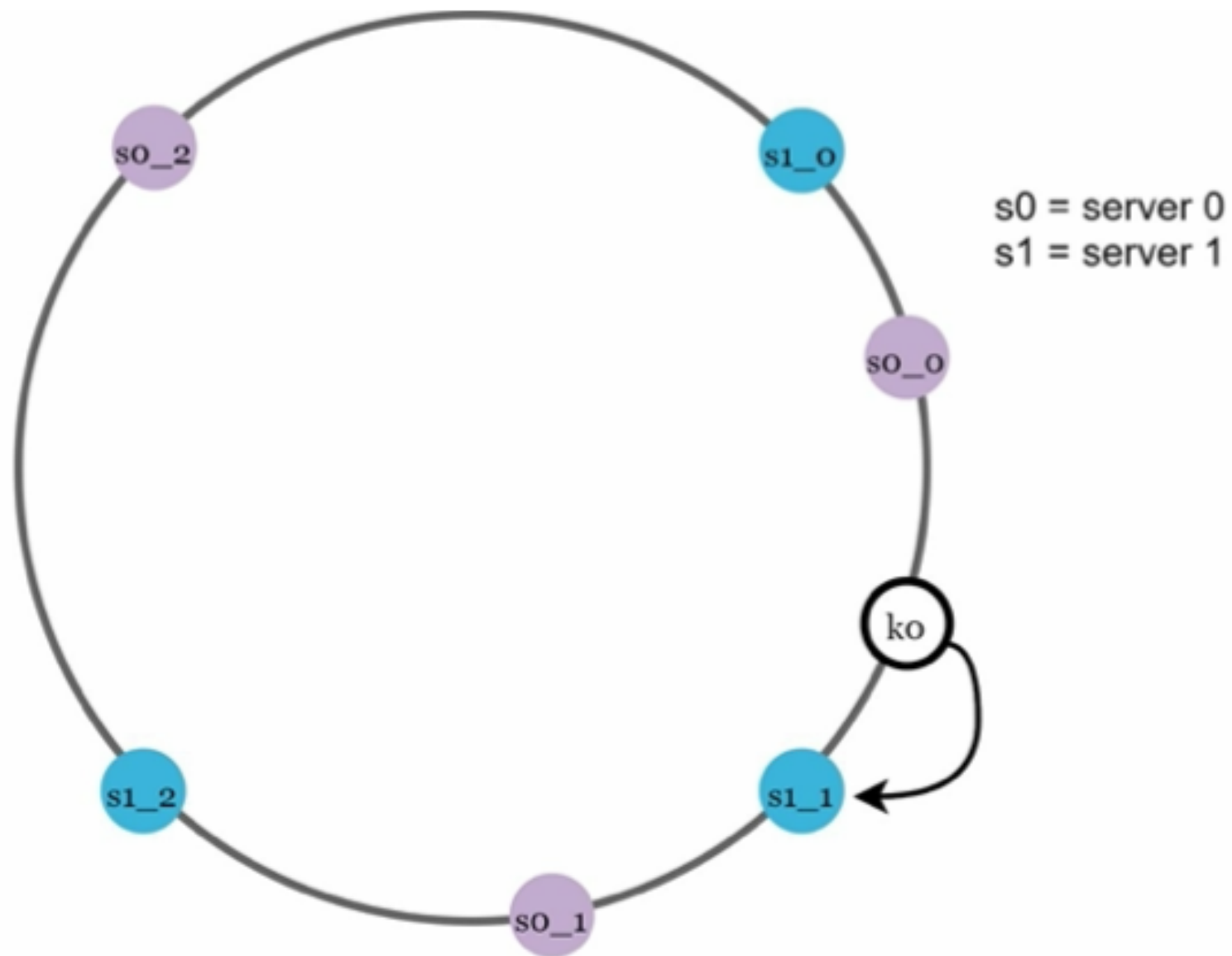
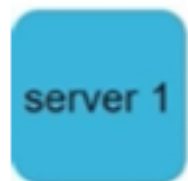


s0 = server 0  
s1 = server 1  
s2 = server 2  
s3 = server 3

Servers



## Servers



## Pros

- ノードが増減した場合に、影響を受けるキーの範囲が限定される
- データの移動が局所的で済む

## Cons

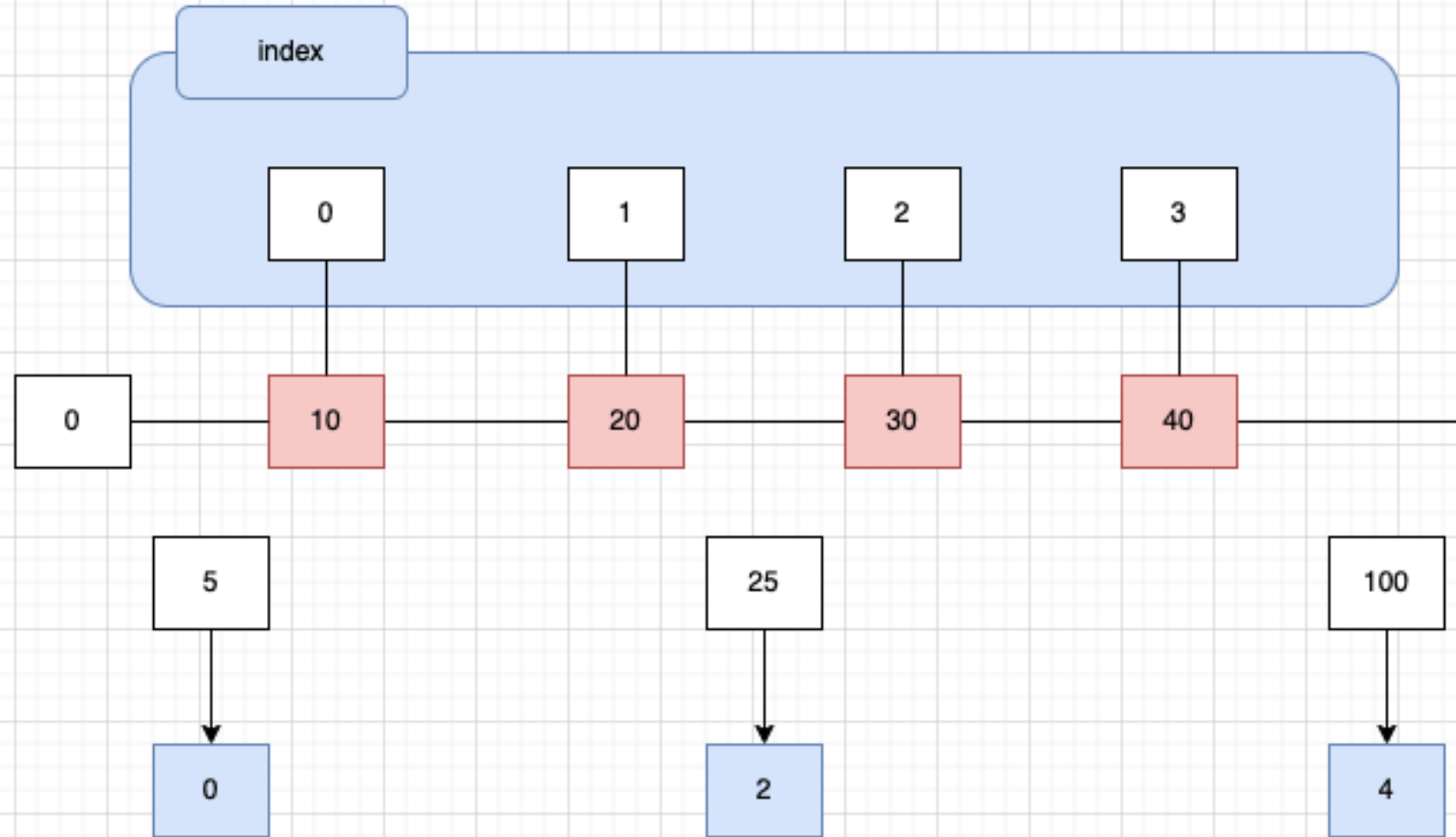
- Node の状態を知っている何かしらは必要
- DB の場合、Node の追加・削除時のデータ配分のロジックが必要(複雑)

## 具体的なコード例

<https://github.com/golang/groupcache/blob/master/consistenthash/consistenthash.go>

sort.Search が便利

# sort.Search





## 参考

- [Consistent Hashing: Algorithmic Tradeoffs](#)
- [System Design Interview – An insider's guide](#)

# 発展

- HOW DISCORD SCALED ELIXIR TO 5,000,000 CONCURRENT USERS
- Shuffle Sharding
- libketama
  - memcached のクライアントライブラリ
    - 今使っているか調べていない
- もう少し実世界よりの実装例
  - node の削除に対応している