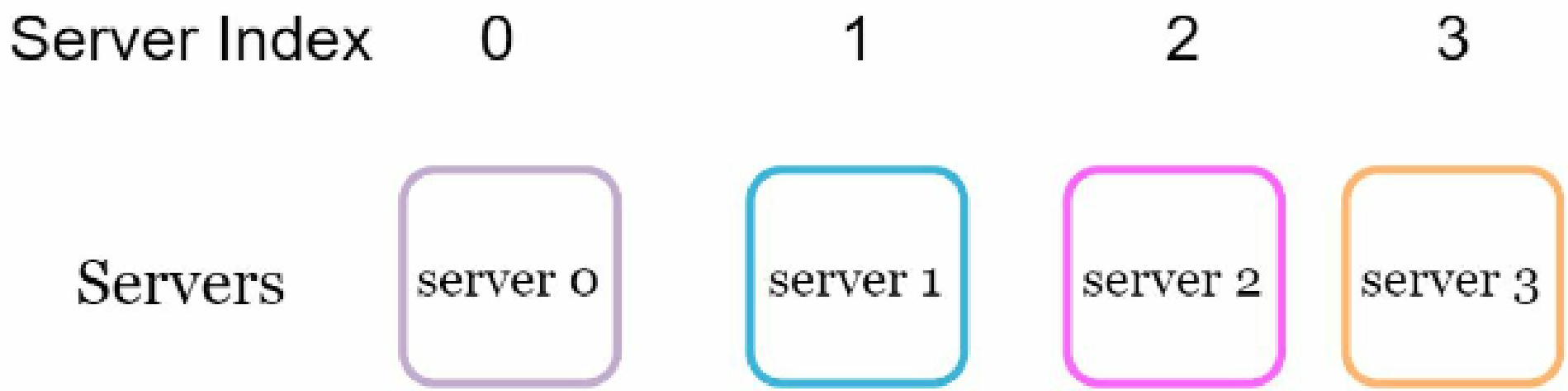
# Consistent Hashing

Kohei Misu

# Overview

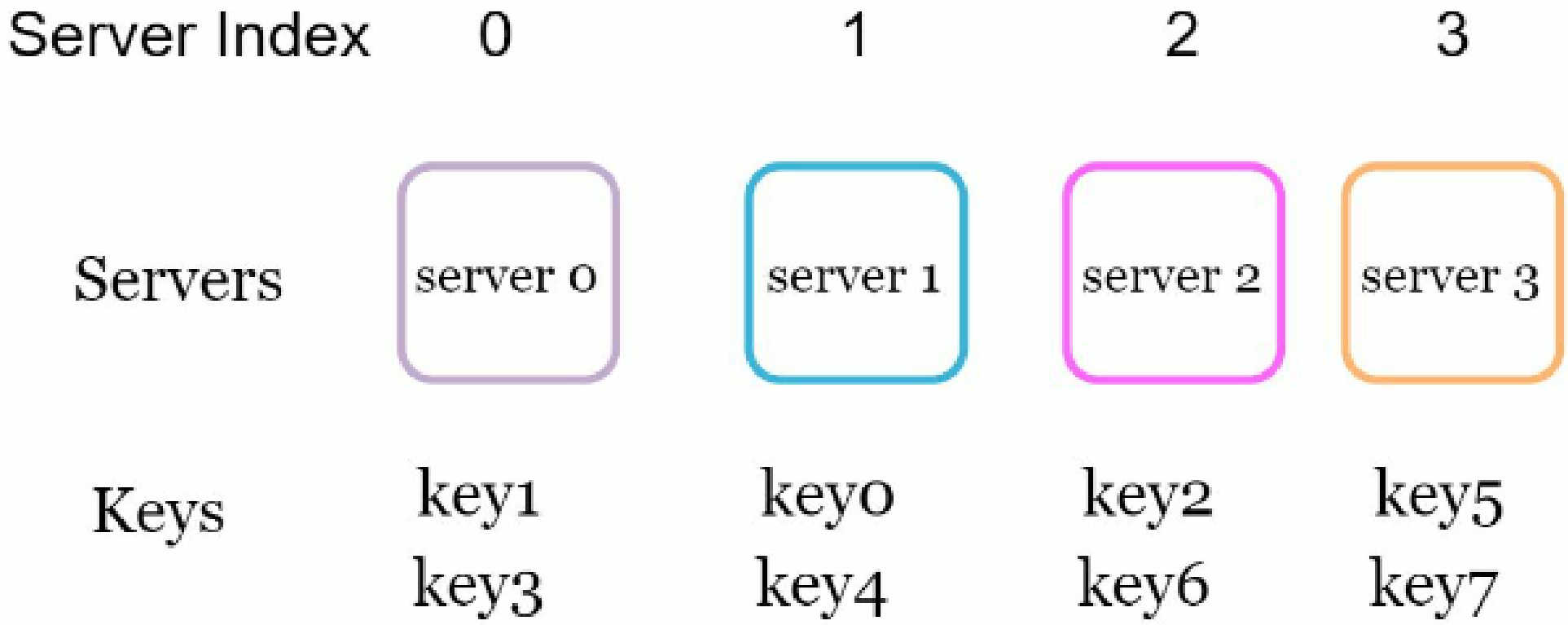
- Rehashing Problem (modular hash)
- Consistent Hashing

## Cache Servers

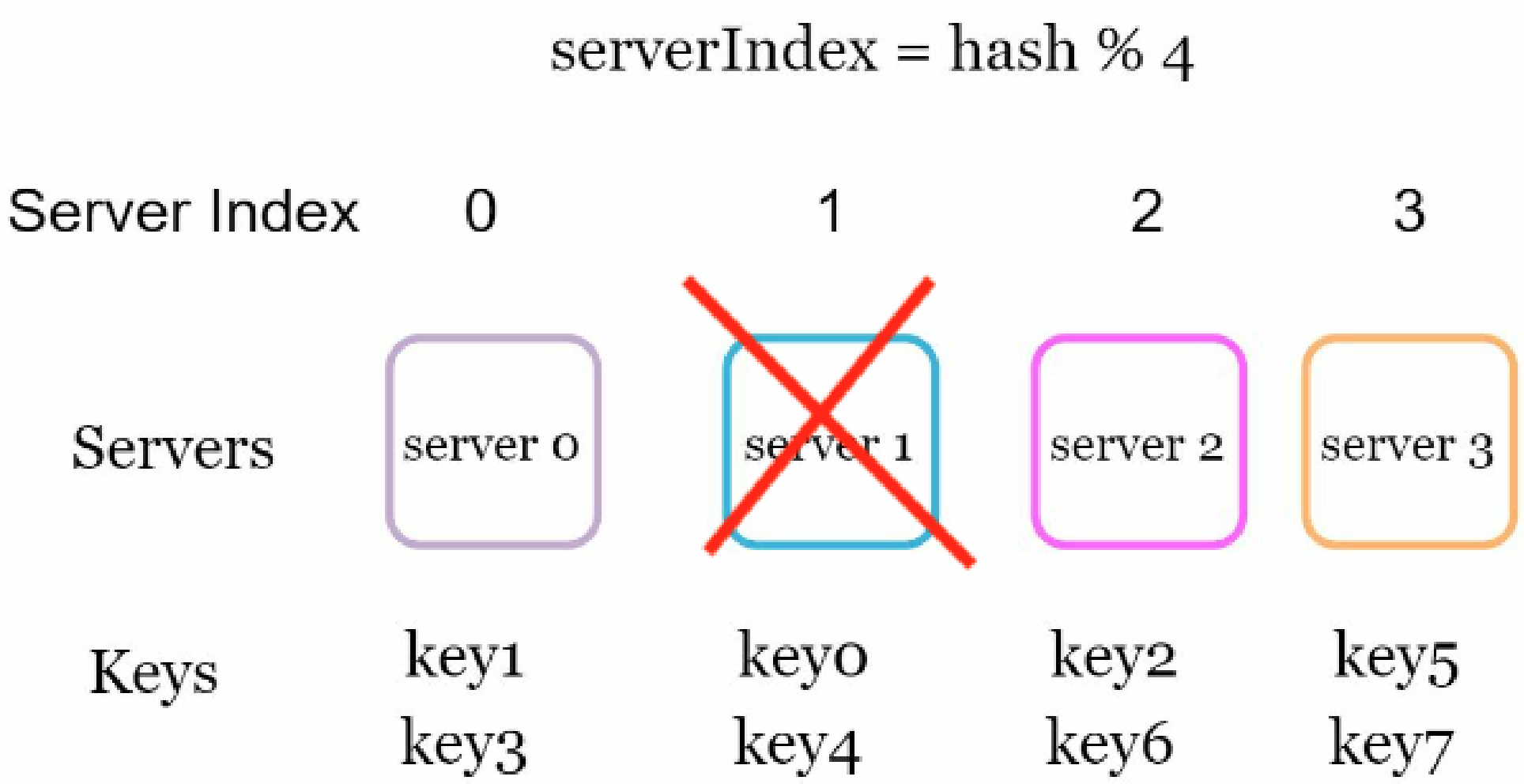


| key | hash | hash % 4 |
|---|---|---|
| key0 f(key0) = | 18358617 | 1 |
| key1 f(key1) = | 26143584 | 0 |
| key2 | 18131146 | 2 |
| key3 | 35863496 | 0 |
| key4 | 34085809 | 1 |
| key5 | 27581703 | 3 |
| key6 | 38164978 | 2 |
| key7 | 22530351 | 3 |

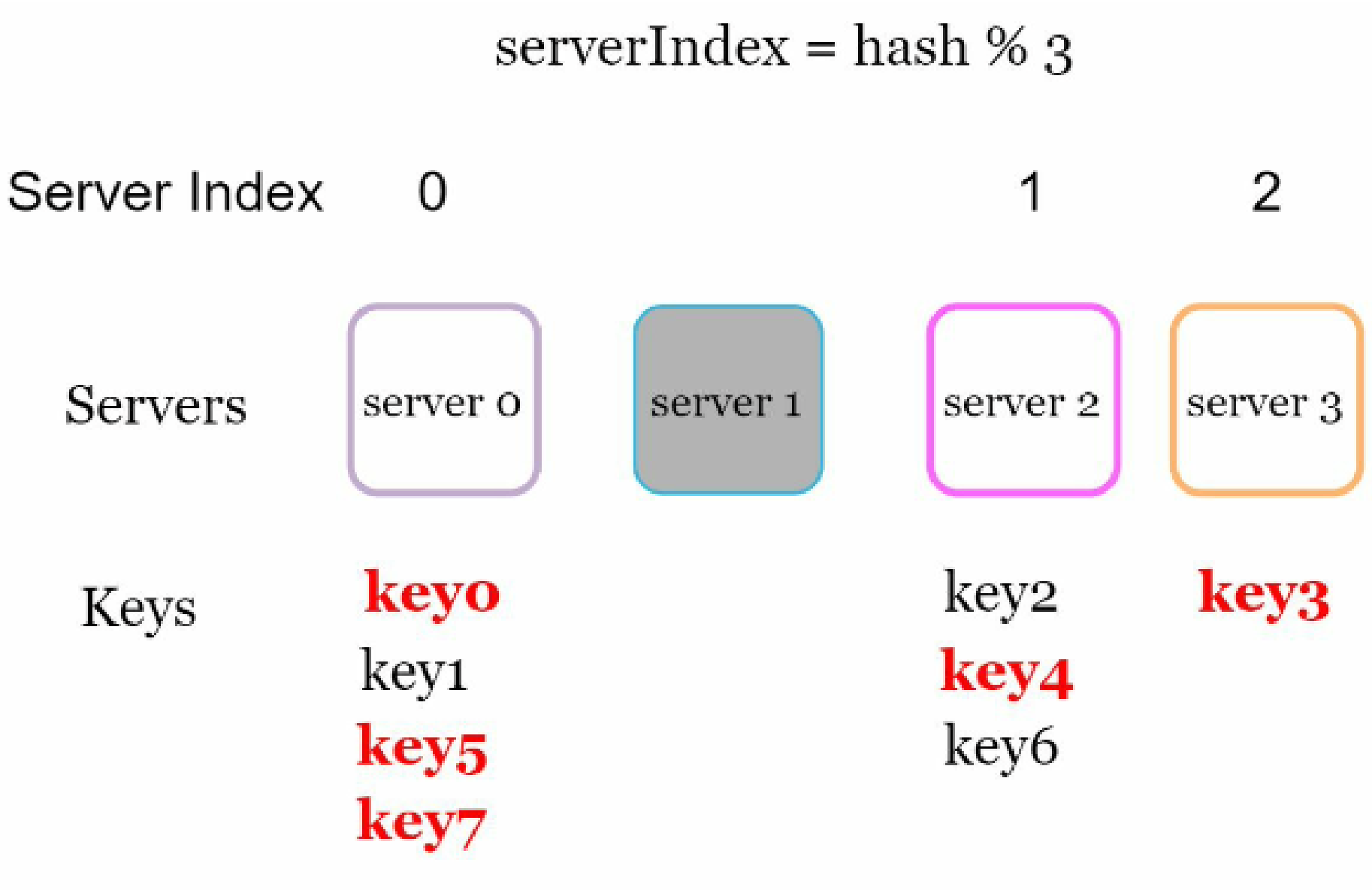
serverIndex = hash % 4



# Server1 が Offline になったとき



| 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 |

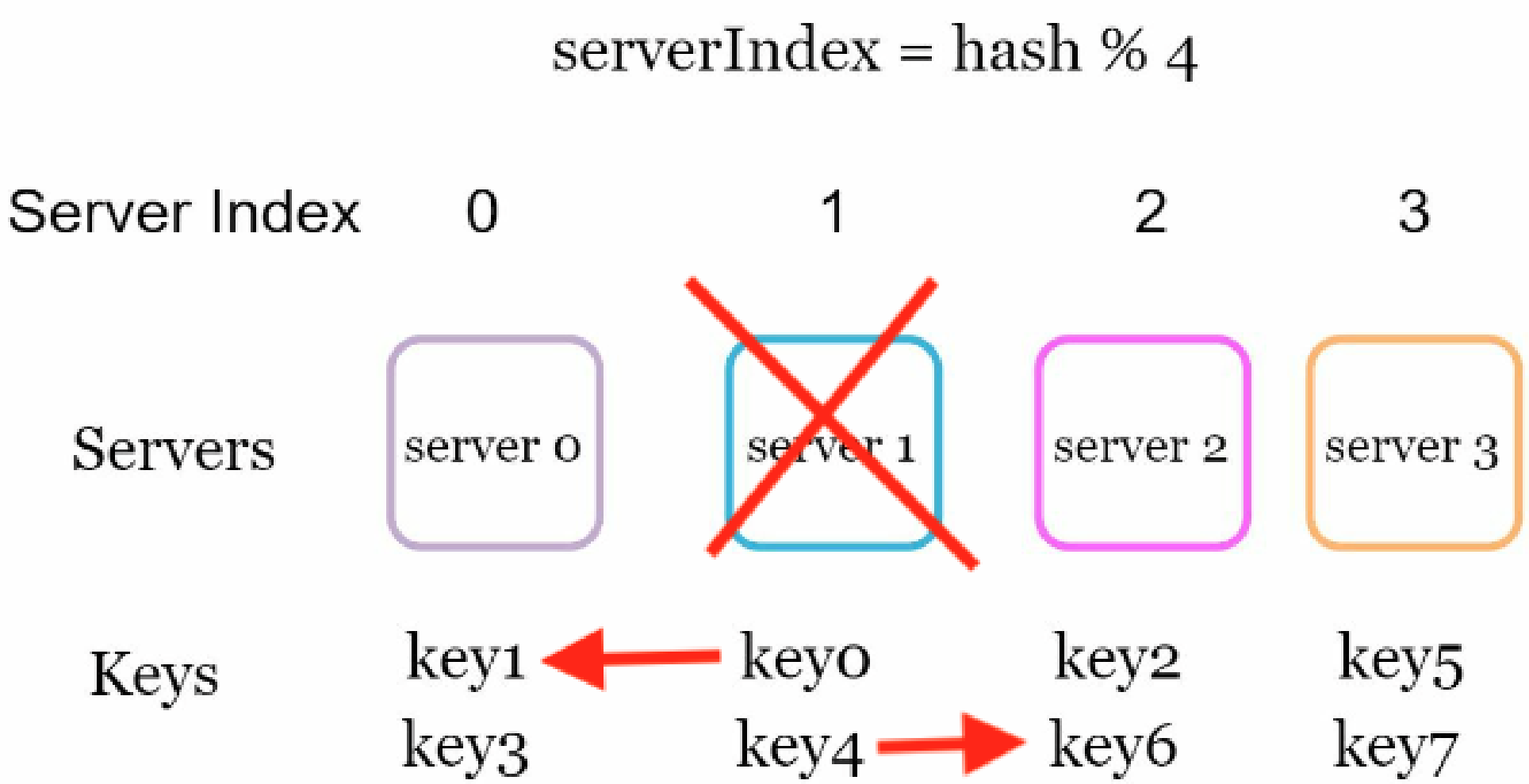

serverIndex = hash % 3
Server Index
0
1
2
Servers
server 0
server 1
server 2
server 3
Keys
key0
key1
key5
key7
key2
key4
key6
key3

# 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

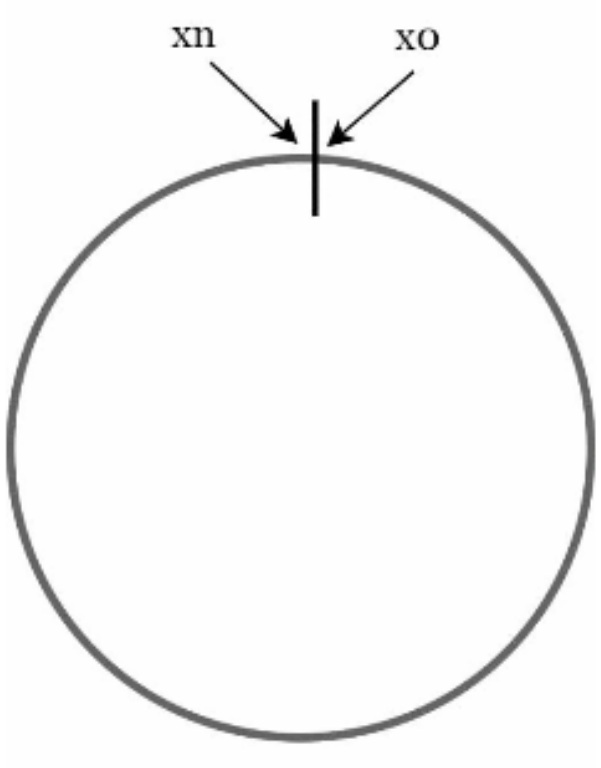
# 理想


serverIndex = hash % 4
Server Index
0
1
2
3
Servers
server 0
server 1
server 2
server 3
Keys
key1
key0
key2
key5
key3
key4
key6
key7

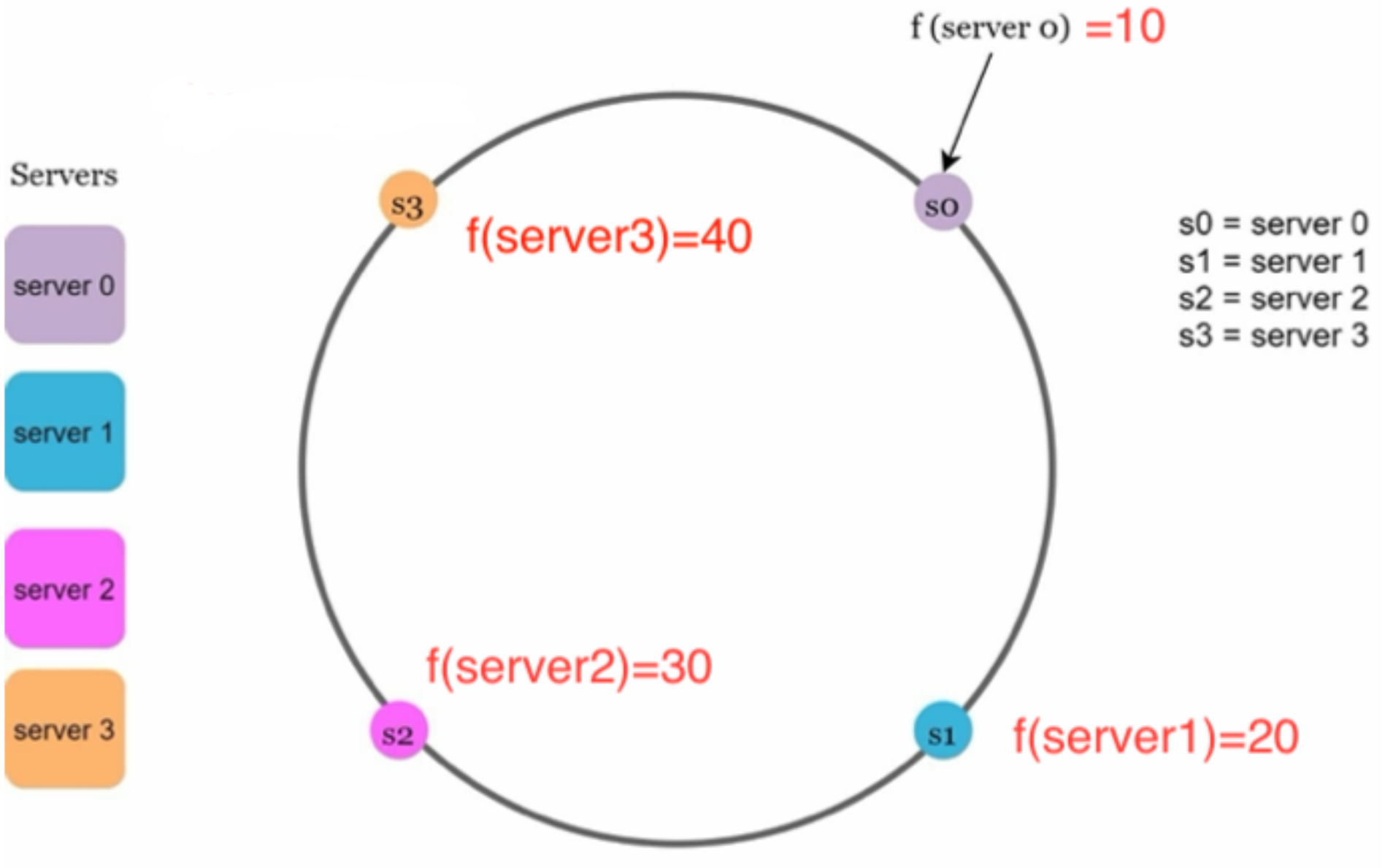
# 実装

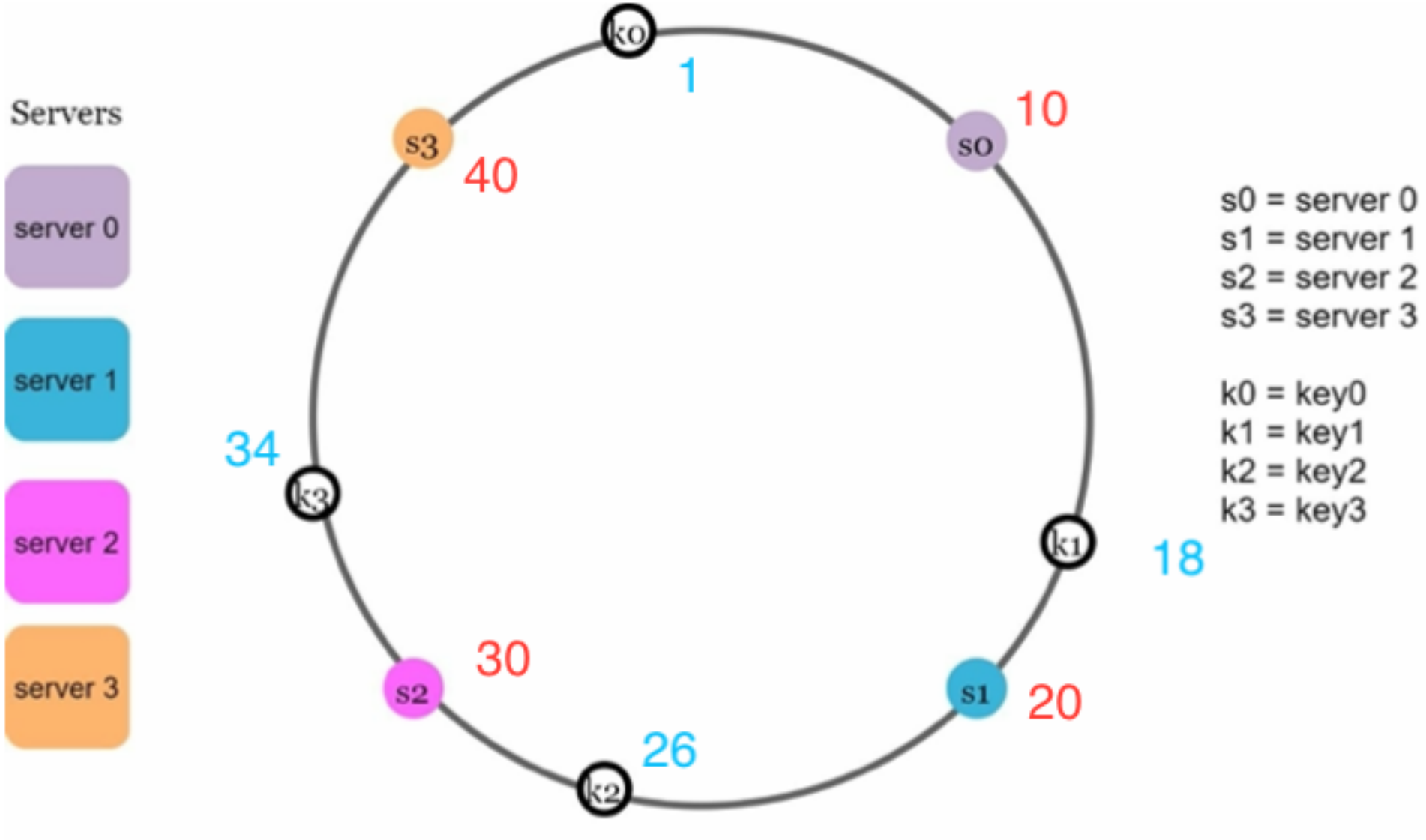
f: hash 関数

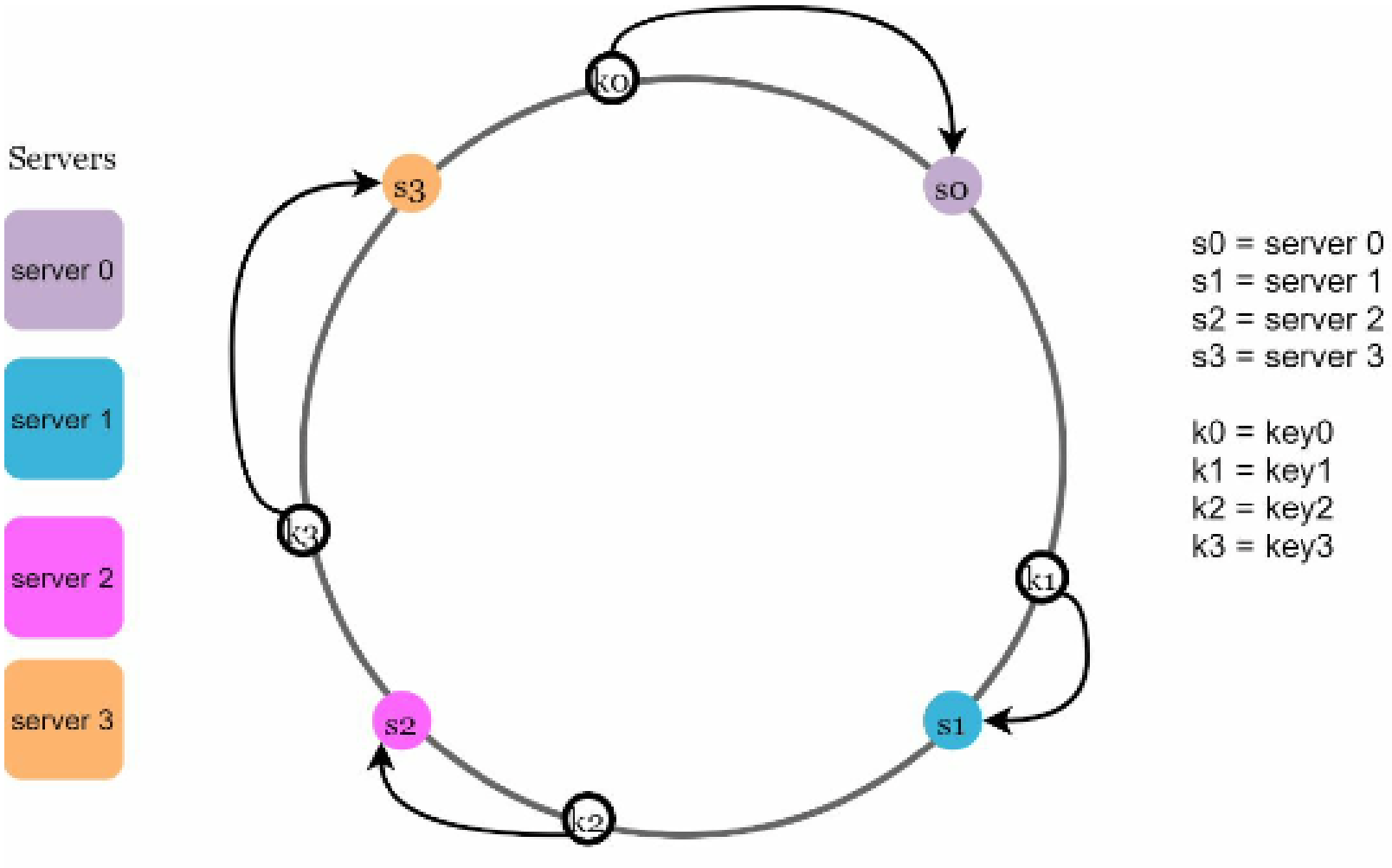
x0, x1 ... xn: 出力範囲

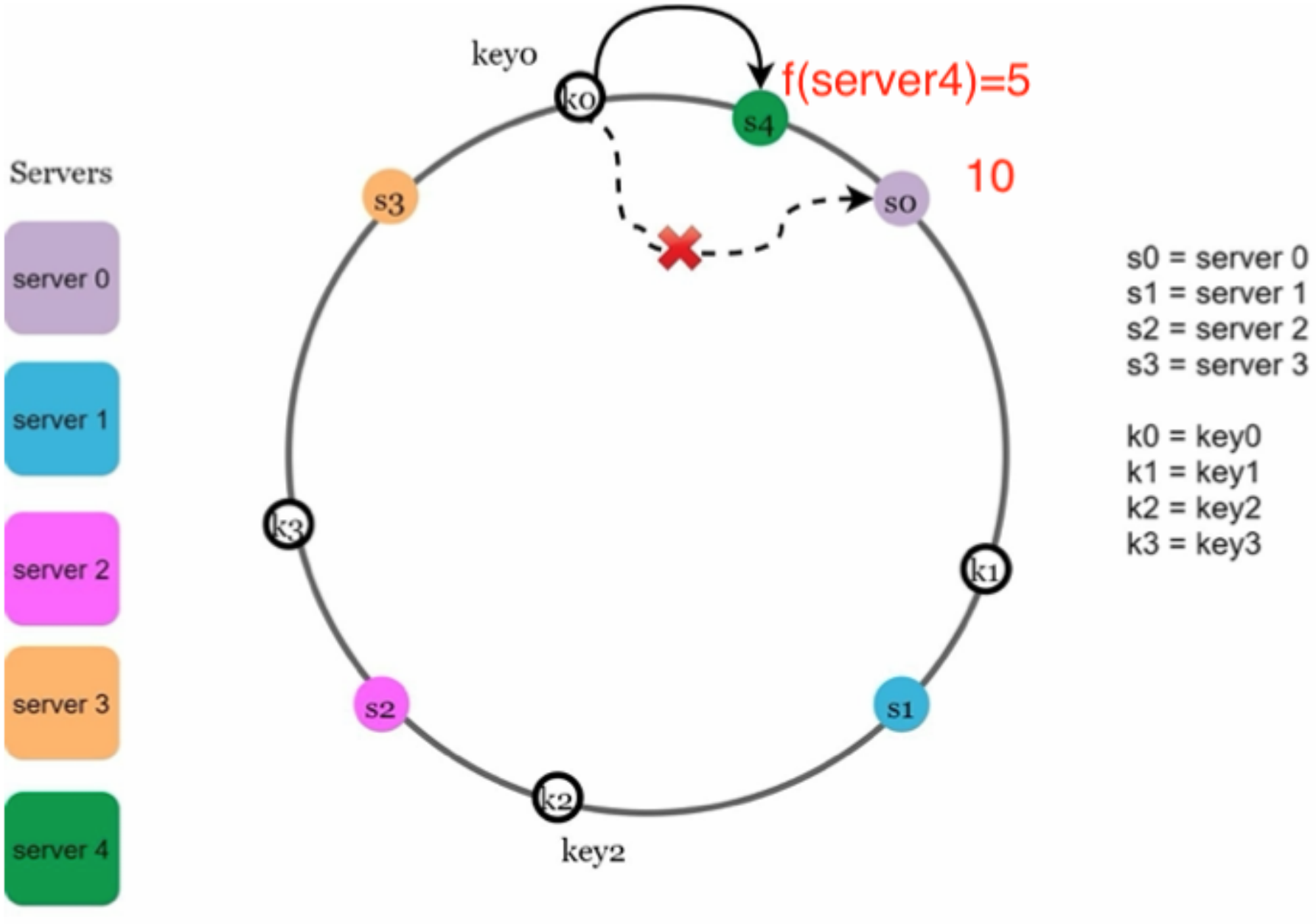


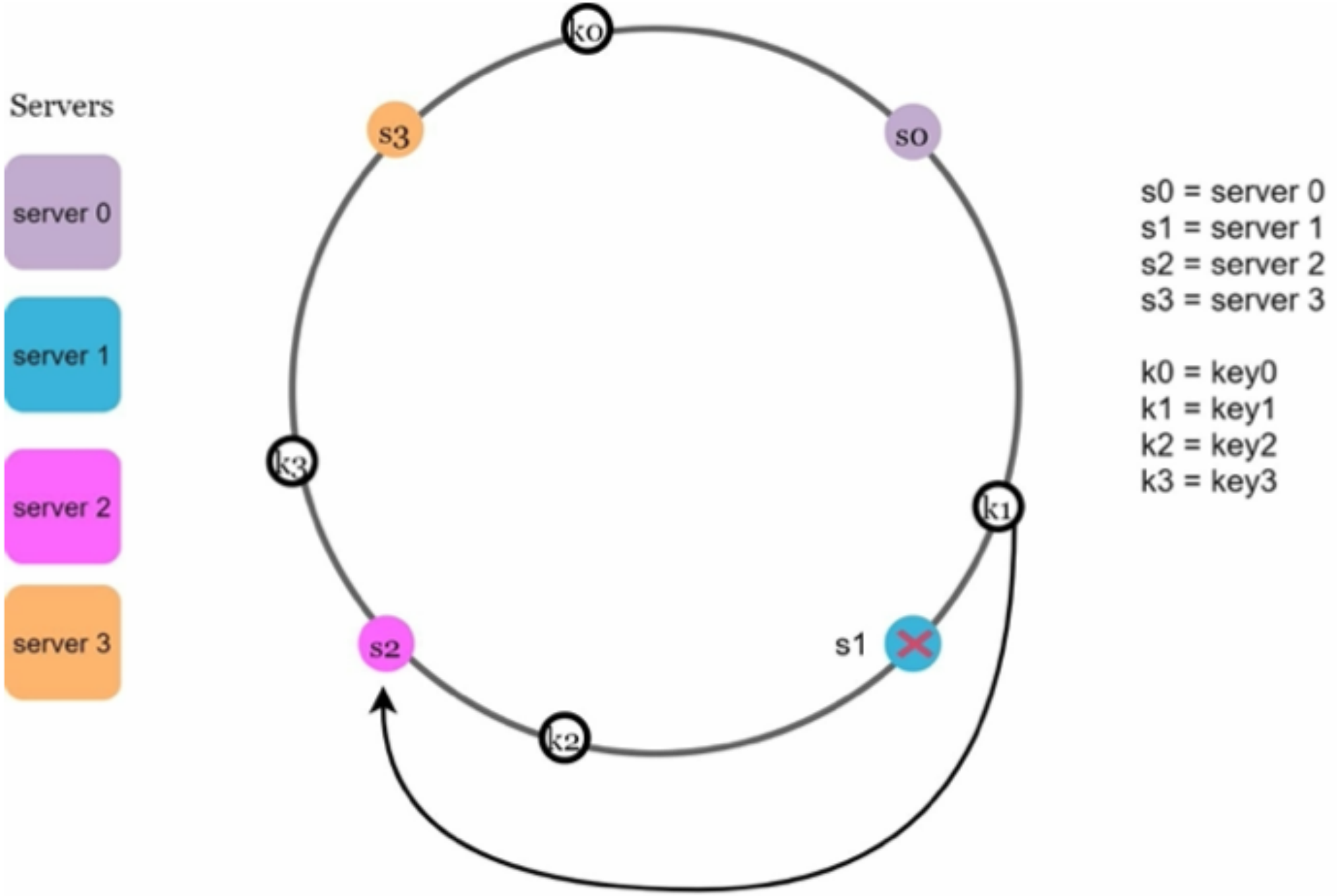

- https://github.com/Cyan4973/xxHash など軽量なものがオススメ

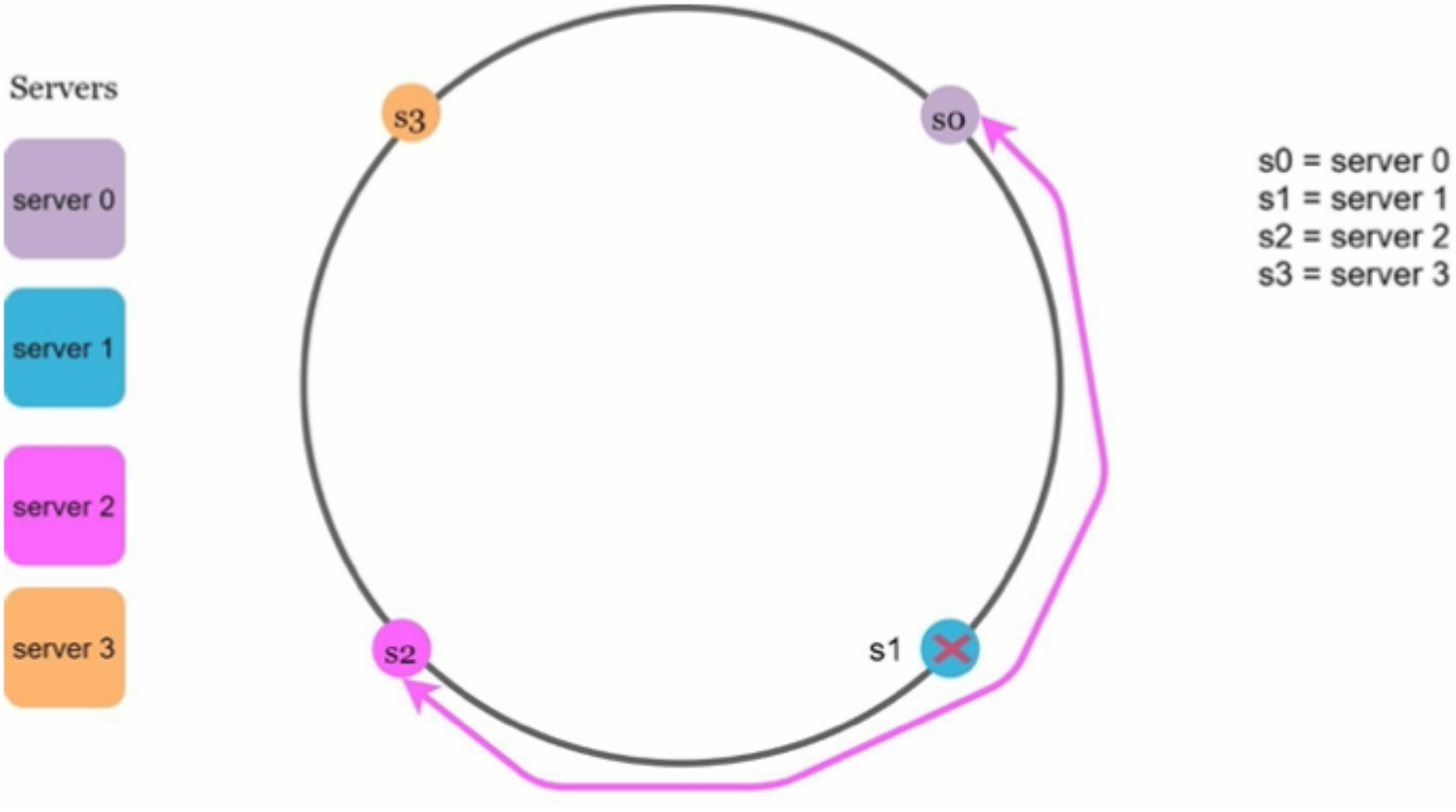

Servers
server 0
server 1
server 2
server 3
f (server o) =10
s3
f(server3)=40
so
s0 = server 0
s1 = server 1
s2 = server 2
s3 = server 3
f(server2)=30
s2
s1
f(server1)=20

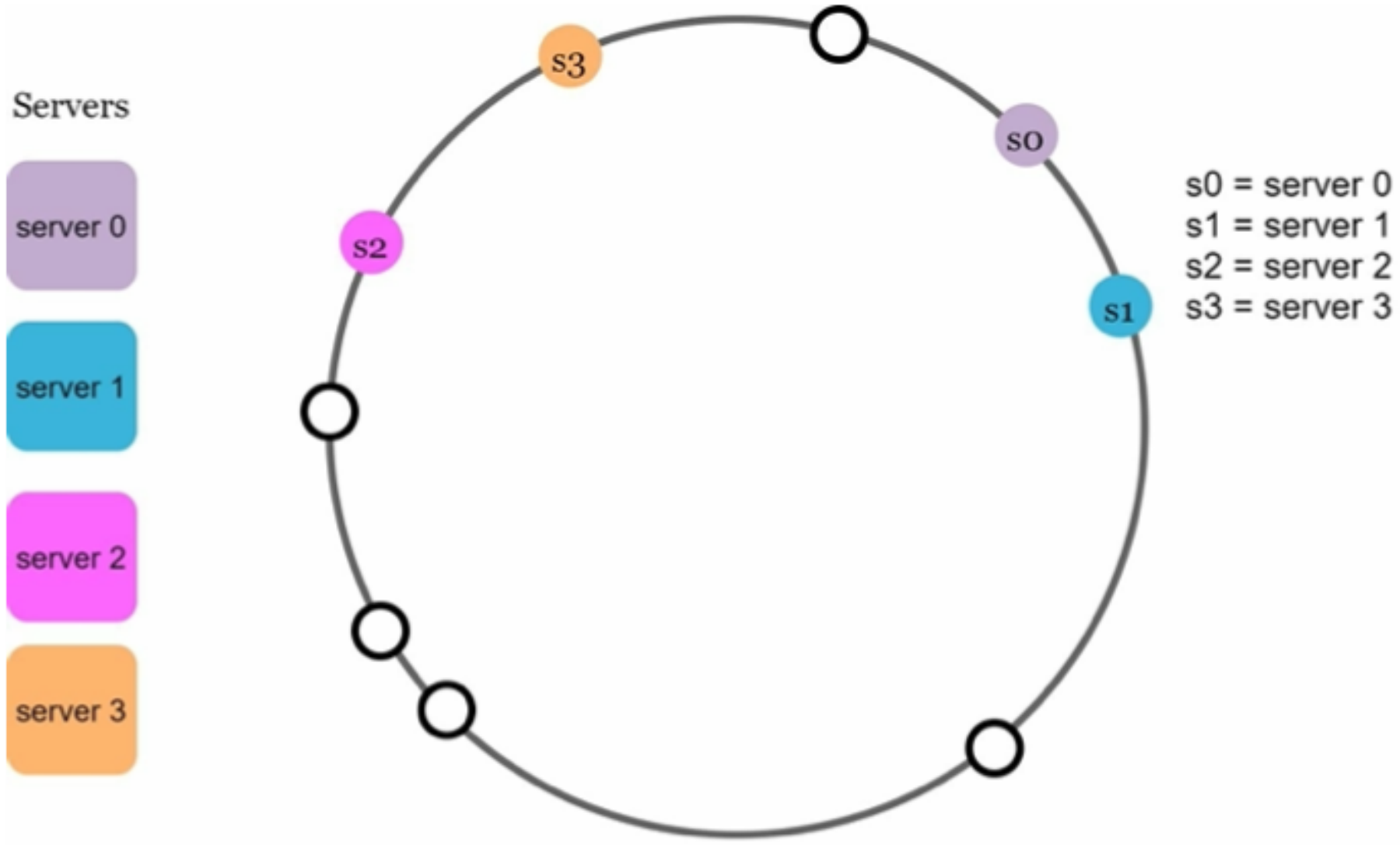

Servers
server 0
server 1
server 2
server 3
k0
1
s0
10
s1
20
k1
18
k2
26
s2
30
k3
34
s3
40
s0 = server 0
s1 = server 1
s2 = server 2
s3 = server 3
k0 = key0
k1 = key1
k2 = key2
k3 = key3

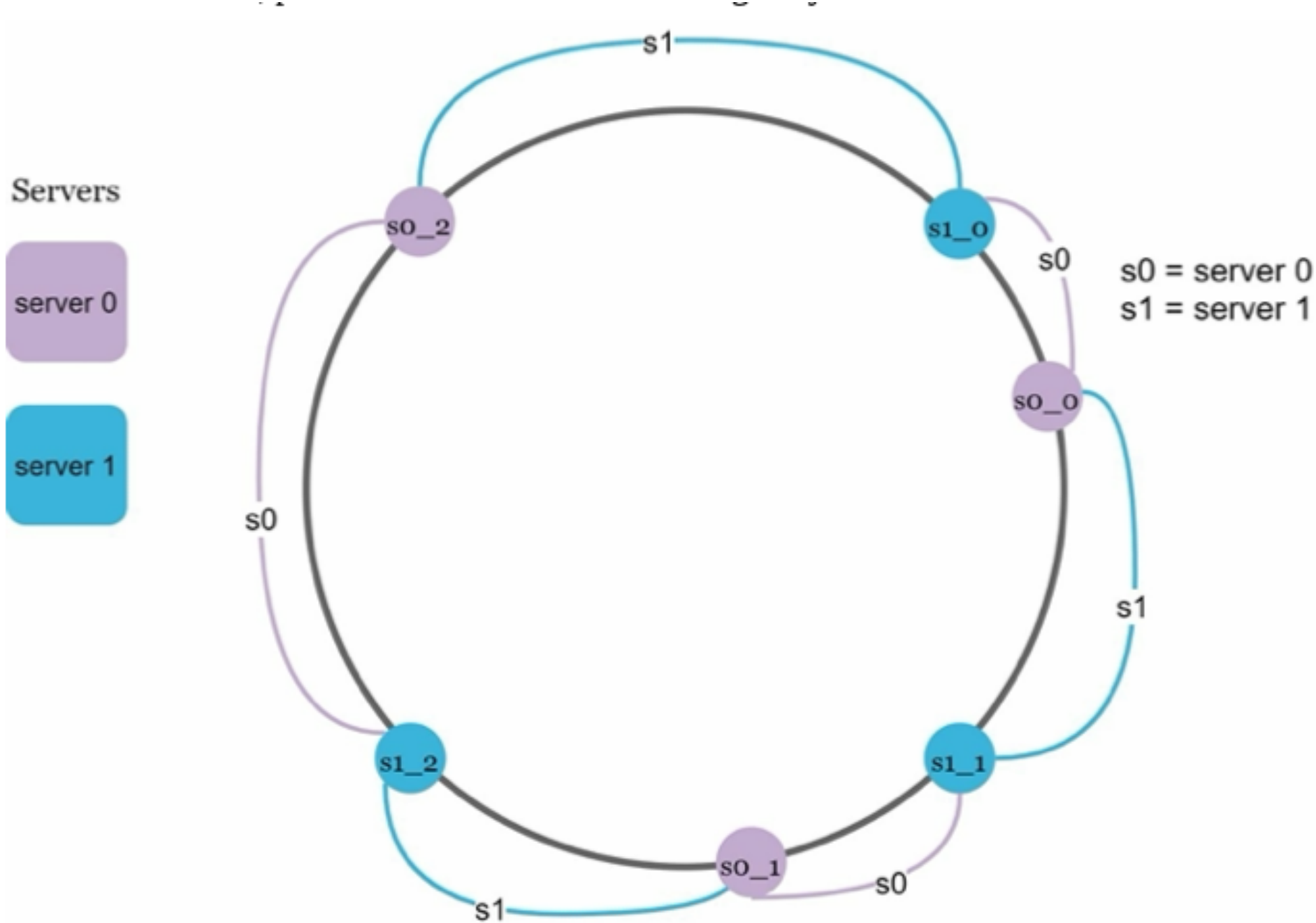

Servers
server 0
server 1
server 2
server 3
k0
s3
s0
k3
k1
s2
s1
k2
s0 = server 0
s1 = server 1
s2 = server 2
s3 = server 3
k0 = key0
k1 = key1
k2 = key2
k3 = key3

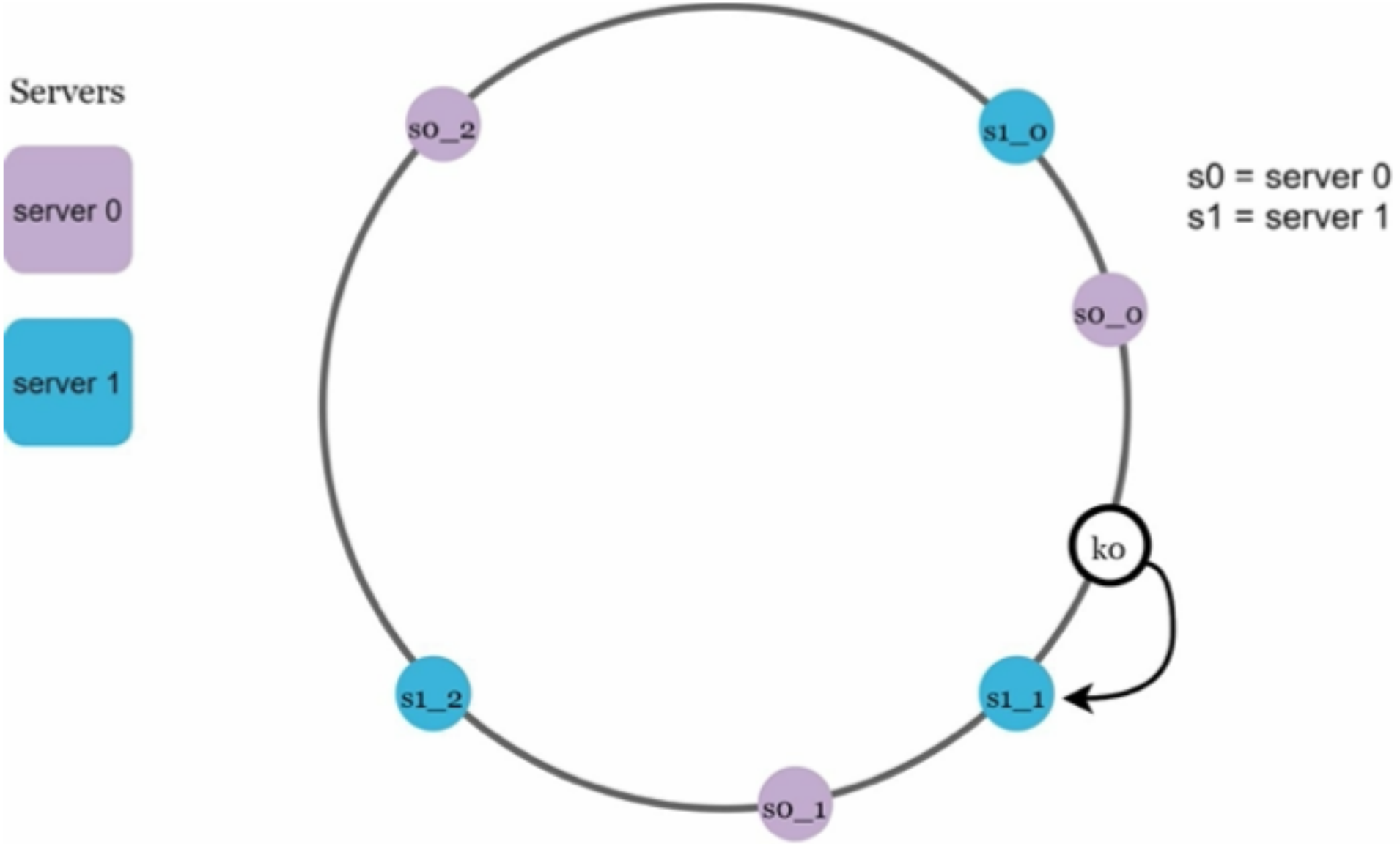

Servers
server 0
server 1
server 2
server 3
server 4
keyo
k0
s4
f(server4)=5
10
s3
s0
k3
k1
s2
s1
k2
key2
s0 = server 0
s1 = server 1
s2 = server 2
s3 = server 3
k0 = key0
k1 = key1
k2 = key2
k3 = key3


Servers
server 0
server 1
server 2
server 3
k0
s3
s0
k3
k1
s2
s1
k2
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
s3
s0
s2
s1
s0 = server 0
s1 = server 1
s2 = server 2
s3 = server 3


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


Servers
server 0
server 1
s1
s0_2
s1_0
s0
s0 = server 0
s1 = server 1
s0_0
s0
s1
s1_2
s1_1
s0_1
s1
s0


Servers
server 0
server 1
s0_2
s1_0
s0_0
k0
s1_1
s0_1
s1_2
s0 = server 0
s1 = server 1

# 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

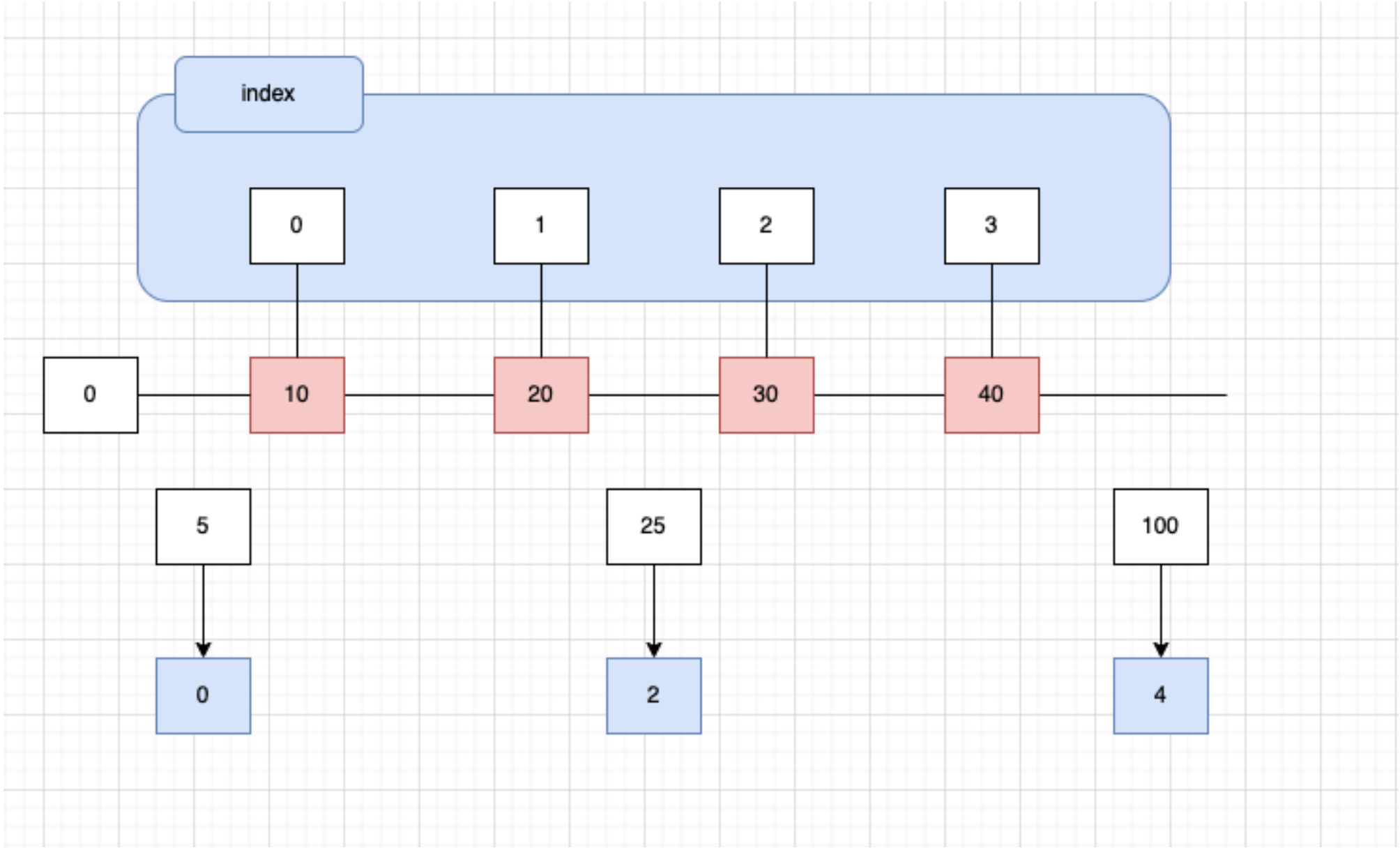
- Node の状態を知っている何かしらは必要
- DB の場合、Node の追加・削除時のデータ配分のロジックが必要(複雑)

# 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