## Kosscon 2019

charsyam@naver.com

# 목차

- Redis 운영
- Redis Cluster
- Redis Failover

## Redis 소개

- In-Memory Data Structure Store
- Open Source(BSD 3 License)
- Support data structures
  - Strings, set, sorted-set, hashes, list
  - Hyperloglog, bitmap, geospatial index
  - Stream
- Only 1 Committer

# Redis 운영

# Redis 운영

- 메모리 관리를 잘하자.
- O(N) 관련 명령어는 주의하자.
- Replication
- 권장 설정 Tip

메모리 관리를 잘하자!

# 메모리 관리

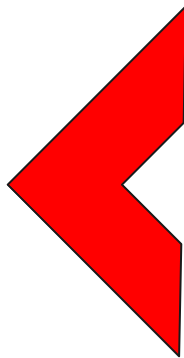
- Redis 는 In-Memory Data Store.
- Physical Memory 이상을 사용하면 문제가 발생
  - Swap 이 있다면 Swap 사용으로 해당 메모리 Page 접근시 마다 늦어짐.
  - Swap이 없다면?
- Maxmemory를 설정하더라도 이보다 더 사용할 가능성이 큼.
- RSS 값을 모니터링 해야함.

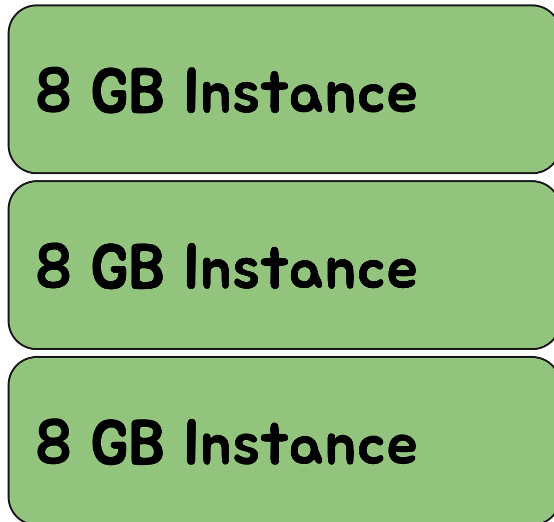
# 메모리 관리

- 많은 업체가 현재 메모리를 사용해서 Swap을 쓰고 있다는 것을 모를때가 많음(T.T)

# 메모리 관리

- 큰 메모리를 사용하는 instance 하나보다는 적은 메모리를 사용하는 instance 여러개가 안전함.
- CPU 4 Core, 32GB Memory





# 메모리 관리

- Redis는 메모리 파편화가 발생할 수 있음. 4.x 대 부터 메모리 파편화를 줄이도록 jemlloc에 힌트를 주는 기능이 들어갔으나, jemalloc 버전에 따라서 다르게 동작할 수 있음.
- 3.x대 버전의 경우
  - 실제 used memory는 2GB로 보고가 되지만 11GB의 RSS를 사용하는 경우가 자주 발생
- 다양한 사이즈를 가지는 데이터 보다는 유사한 크기의 데이터를 가지는 경우가 유리.

# 메모리가 부족할 때는?

- Cache is Cash!!!
  - 좀 더 메모리 많은 장비로 Migration.
  - 메모리가 빡빡하면 Migration 중에 문제가 발생할수도...
- 있는 데이터 줄이기.
  - 데이터를 일정 수준에서만 사용하도록 특정 데이터를 줄임.
  - 다만 이미 Swap을 사용중이라면, 프로세스를 재시작 해야함.

# 메모리를 줄이기 위한 설정

- 기본적으로 Collection 들은 다음과 같은 자료구조를 사용
  - Hash -> HashTable을 하나 더 사용
  - Sorted Set -> Skiplist와 HashTable을 이용.
  - Set -> HashTable 사용
  - 해당 자료구조들은 메모리를 많이 사용함.
- Ziplist 를 이용하자.

# Ziplist 구조

- In-Memory 특성 상, 적은 개수라면 선형 탐색을 하더라도 빠르다.

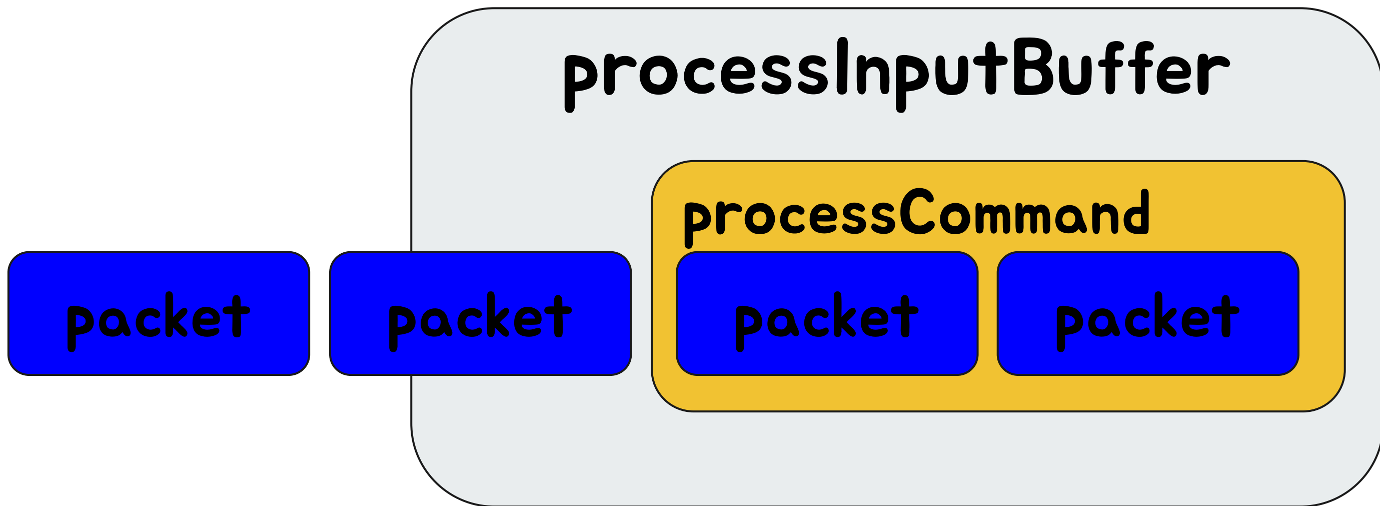
- List, hash, sorted set 등을 ziplist로 대체해서 처리를 하는 설정이 존재
  - hash-max-ziplist-entries, hash-max-ziplist-value
  - list-max-ziplist-size, list-max-ziplist-value
  - zset-max-ziplist-entries, zset-max-ziplist-value

O(N) 관련 명령어는 주의하자.

# O(N) 관련 명령어는 주의하자.

- Redis 는 Single Threaded.
  - 그러면 Redis가 동시에 여러 개의 명령을 처리할 수 있을까?
  - 참고로 단순한 get/set의 경우, 초당 10만 TPS 이상 가능(CPU속도에 영향을 받습니다.)

# Redis is Single Threaded.




Packet 으로 하나의 Command가 완성되면
processCommand 에서 실제로 실행됨

## Single Threaded 의 의미

- Redis 는 Single Threaded.
  - 그러면 Redis가 동시에 여러 개의 명령을 처리할 수 있을까?
  - 참고로 단순한 get/set의 경우, 초당 10만 TPS 이상 가능(물리 서버기준, CPU속도에 영향을 받습니다.)

## Single Threaded 의 의미

- 한번에 하나의 명령만 수행 가능
  - 그럼 긴 시간이 필요한 명령을 수행하면?

# 망합니다!!!

# 대표적인 O(N) 명령들

- KEYS
- FLUSHALL, FLUSHDB
- Delete Collections
- Get All Collections

# 대표적인 실수 사례

- Key가 백만개 이상인데 확인을 위해 KEYS 명령을 사용하는 경우
  - 모니터링 스크립트가 일초에 한번씩 keys를 호출하는 경우도...
- 아이템이 몇만개 든 hash, sorted set, set에서 모든 데이터를 가져오는 경우
- 예전의 Spring security oauth RedisTokenStore

# KEYS 는 어떻게 대체할 것인가?

- scan 명령을 사용하는 것으로 하나의 긴 명령을 짧은 여러번의 명령으로 바꿀 수 있다.

```
redis 127.0.0.1:6379> scan 0
1) "17"
2)  1) "key:12"
    2) "key:8"
    3) "key:4"
redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18"
   3) "key:0"
   7) "key:6"
   8) "key:9"
   9) "key:11"
```

## Collection의 모든 item을 가져와야 할 때?

- Collection의 일부만 가져오거나…
  - Sorted Set
- 큰 Collection을 작은 여러개의 Collection으로 나눠서 저장
  - Userranks -> Userrank1, Userrank2, Userrank3
  - 하나당 몇천개 안쪽으로 저장하는게 좋음.

# Spring security oauth RedisTokenStore 이슈

- Access Token의 저장을 List(O(N)) 자료구조를 통해서 이루어짐.
  - 검색, 삭제시에 모든 item을 매번 찾아봐야 함.
    - 100만개 쯤 되면 전체 성능에 영향을 줌.
  - 현재는 Set(O(1))을 이용해서 검색, 삭제를 하도록 수정되어있음.

# Spring security oauth RedisTokenStore 이슈

- https://charsyam.wordpress.com/2018/05/11/입-개발-spring-security-oauth의-redistokenstore의-사용은-서비스에-적합하지-않/

charsyam 1:02 pm on May 11, 2018

## [입 개발] spring-security-oauth의 RedisTokenStore의 사용은 서비스에 적합하지 않습니다.

해당 내용은 spring-security-oauth 가 패치되면서, 현재는 성능적인 이슈는 해결된 상황입니다. https://github.com/spring-projects/spring-security-oauth/commit/60f39ce82f380299cb1894baa02d65606f8f1365 다만 Redis를 TokenStore로 사용하면 여전히 메모리 관리에는 신경을 쓰셔야 합니다.

안녕하세요. 입개발 CharSyam 입니다. 저의 대부분의 얘기는 한귀로 듣고 한귀를 씻으시면 됩니다.(엉?) 일단 제목만 보면, 많은 Spring 유저들에게, 저넘의 입개발, 스프링도 모르면서라는 이야기를 들을듯 합니다.(아, 아이돌 까던 분들이 집단 따돌림을 당할 때의 느낌을 미리 체험할 수 있을듯 합니다. – 강해야 클릭율이 올라가는!!!)

먼저, 내용을 시작하기 전에 저는 Spring 맹, Java 맹으로 무식하다는 걸 미리 공개하고 넘어갑니다. 흑흑흑 (나의 봄님이 이럴 리 없어!!!)

Redis Replication

# Redis Replication

- Async Replication
  - Replication Lag 이 발생할 수 있다.
- 'Replicaof'(>= 5.0.0) or 'slaveof' 명령으로 설정 가능
  - Replicaof hostname port
- DBMS로 보면 statement replication가 유사

# Redis Replication

- Replication 설정 과정
    - Secondary에 replicaof or slaveof 명령을 전달
    - Secondary는 Primary에 sync 명령 전달
    - Primary는 현재 메모리 상태를 저장하기 위해
        - Fork
    - Fork 한 프로세서는 현재 메모리 정보를 disk에 dump
    - 해당 정보를 secondary 에 전달
    - Fork 이후의 데이터를 secondary에 계속 전달

## Redis Replication 시 주의할 점

- Replication 과정에서 fork 가 발생하므로 메모리 부족이 발생할 수 있다.
- Redis-cli --rdb 명령은 현재 상태의 메모리 스냅샷을 가져오므로 같은 문제를 발생시킴
- AWS나 클라우드의 Redis는 좀 다르게 구현되어서 좀더 해당 부분이 안정적.

# Redis Replication 시 주의할 점

- 많은 대수의 Redis 서버가 Replica를 두고 있다면
  - 네트웍 이슈나, 사람의 작업으로 동시에 replication이 재시도 되도록 하면 문제가 발생할 수 있음.
  - ex) 같은 네트웍안에서 30GB를 쓰는 Redis Master 100대 정도가 리플리케이션을 동시에 재시작하면 어떤 일이 벌어질 수 있을까?

권장 설정 Tip(redis.conf)

## redis.conf 권장 설정 Tip

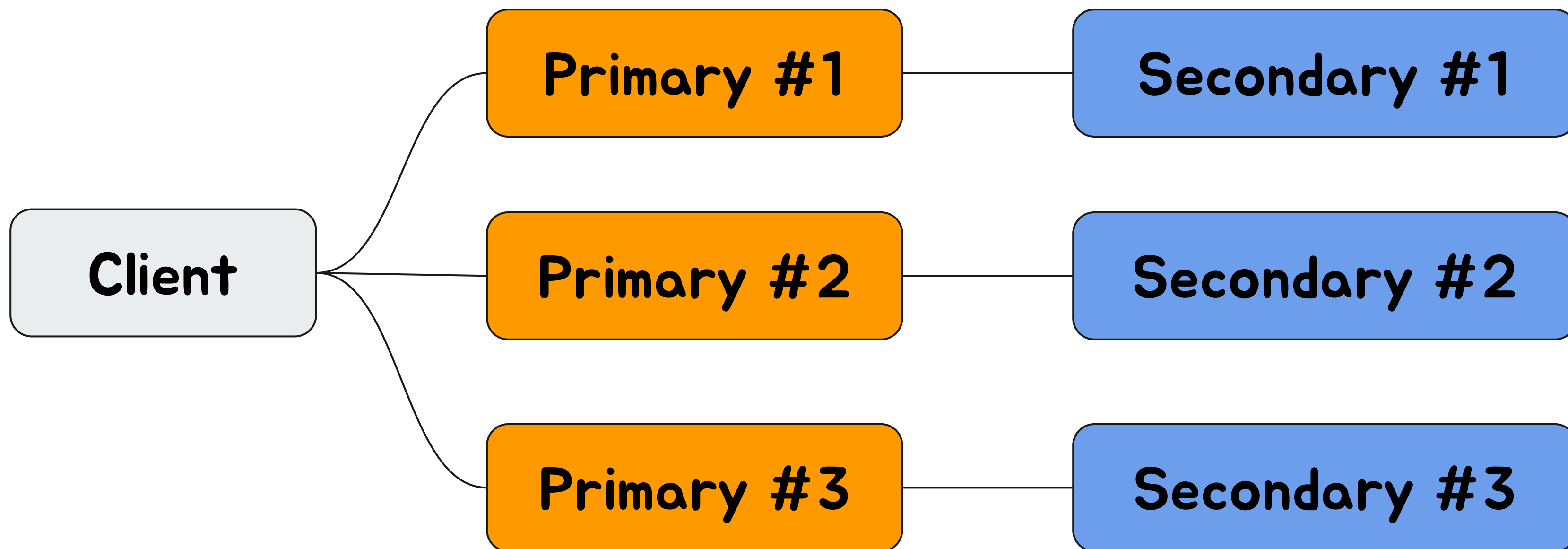
- Maxclient 설정 50000
- RDB/AOF 설정 off
- 특정 commands disable
  - Keys
  - AWS의 ElasticCache는 이미 하고 있음.
- 전체 장애의 90% 이상이 KEYS와 SAVE 설정을 사용해서 발생.
- 적절한 ziplist 설정

# Redis Cluster

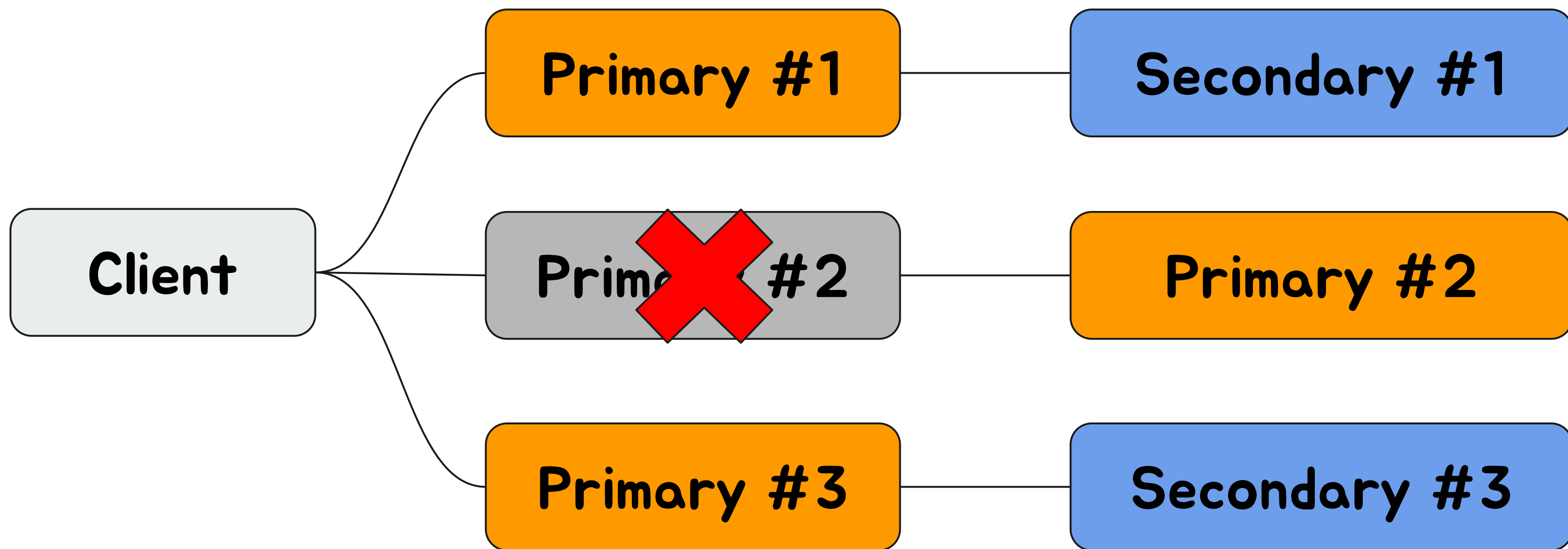
## Redis Cluster

- Hash 기반으로 Slot 16384 로 구분
  - Hash 알고리즘은 CRC16을 사용
  - Slot = crc16(key) % 16384
  - Key가 Key{hashkey} 패턴이면 실제 crc16에 hashkey가 사용된다.
  - 특정 Redis 서버는 이 slot range를 가지고 있고, 데이터 migration은 이 slot 단위의 데이터를 다른 서버로 전달하게 된다.(migrateCommand 이용)

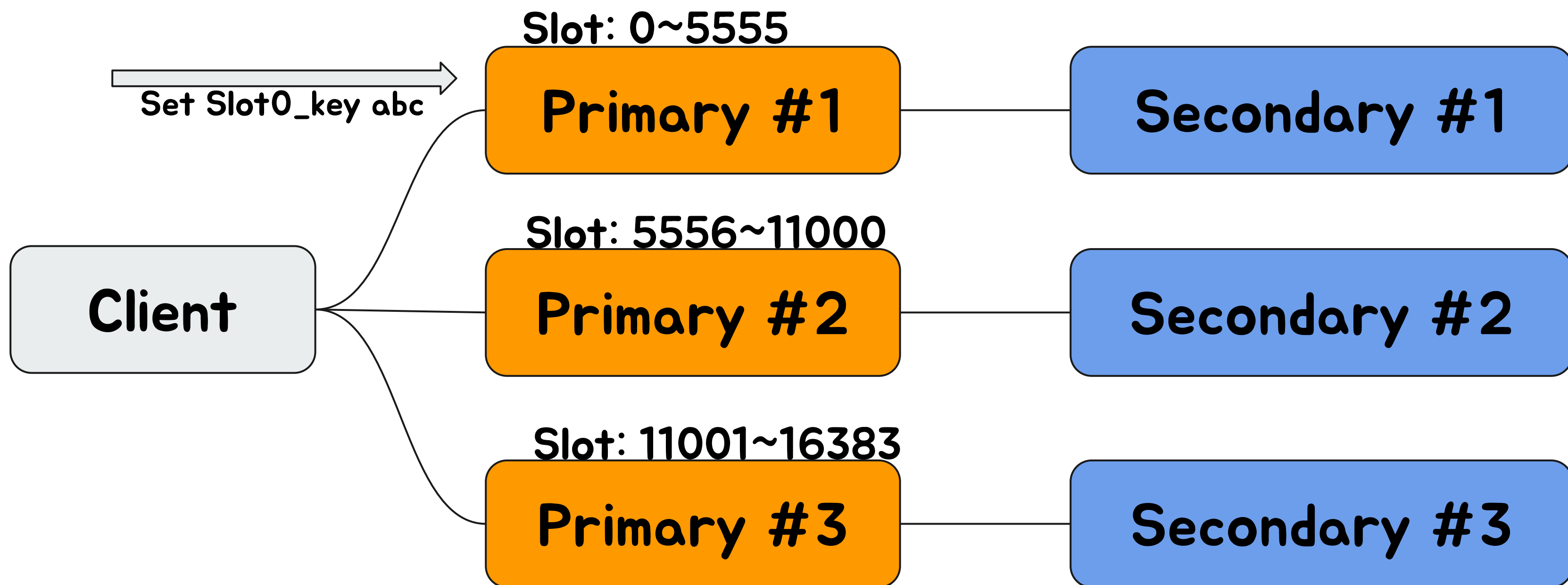
# Redis Cluster



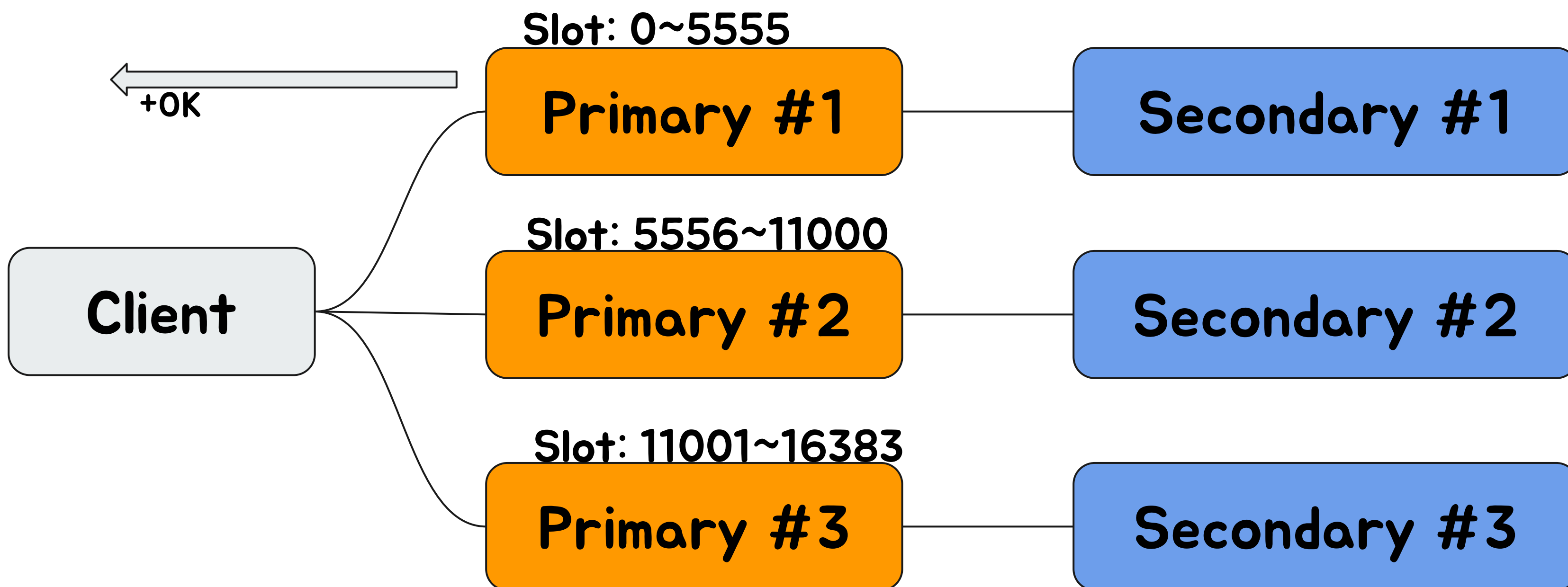
# Redis Cluster



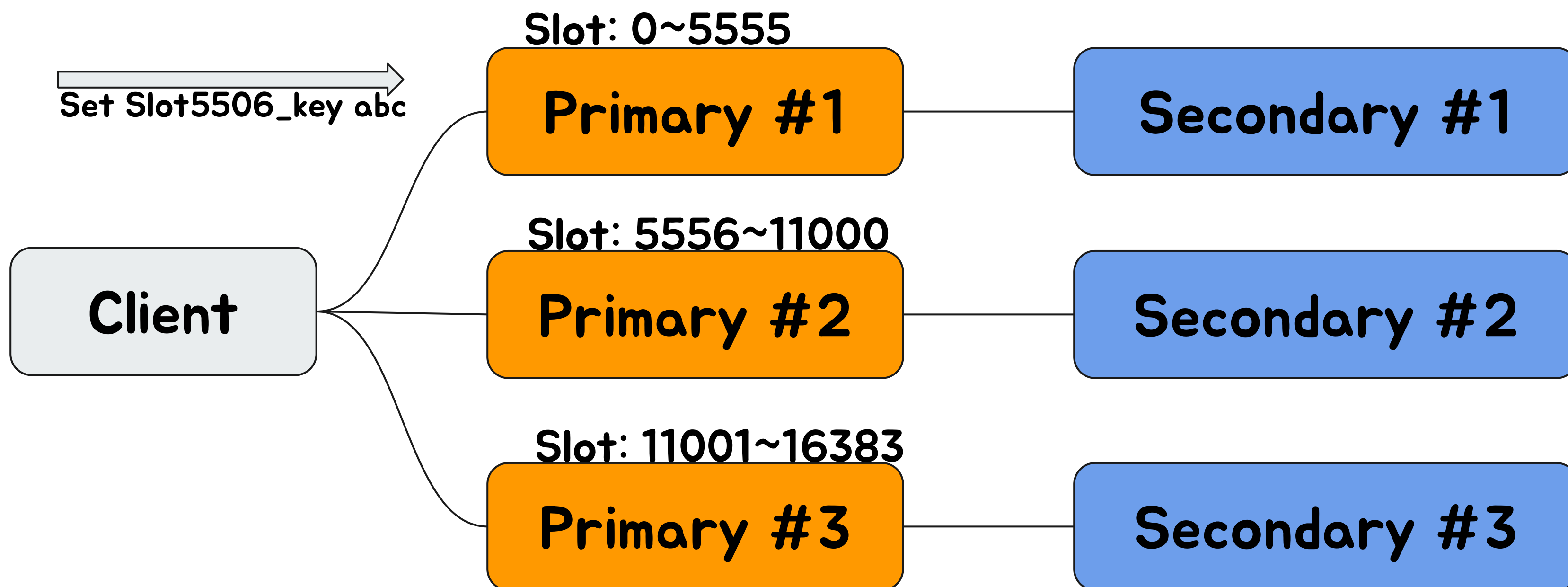
# Redis Cluster



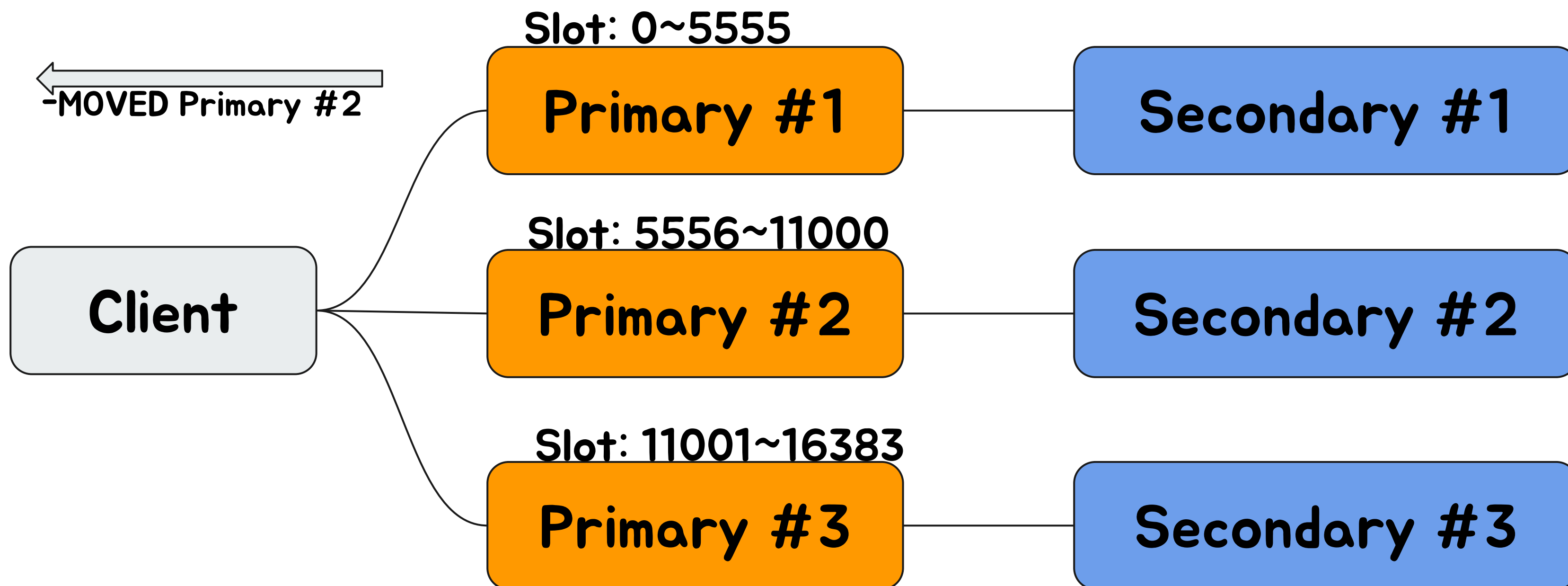
# Redis Cluster



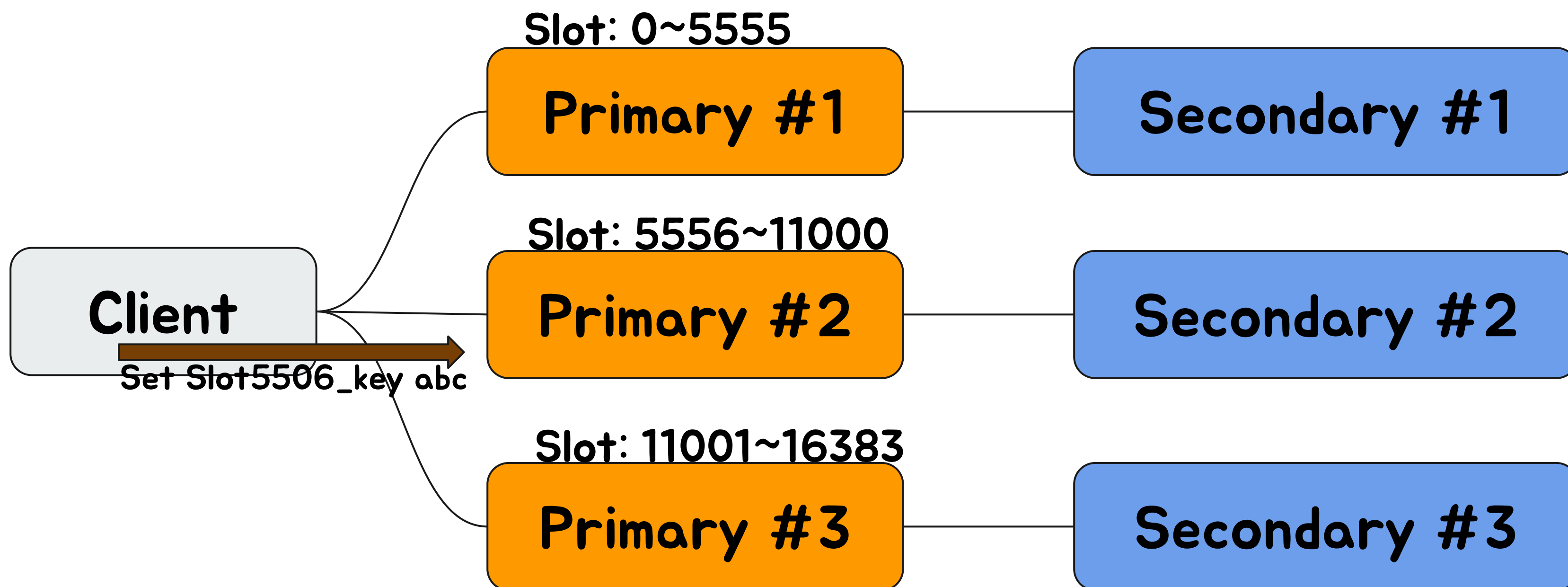
# Redis Cluster



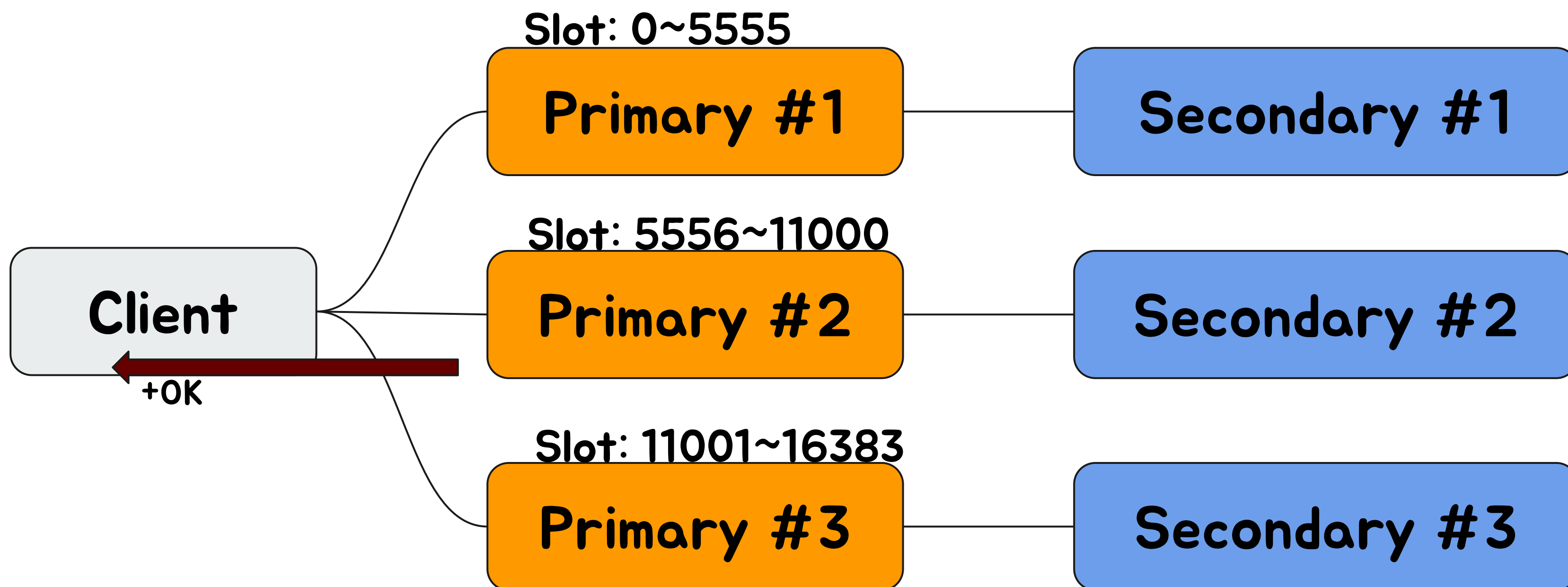
# Redis Cluster



# Redis Cluster



# Redis Cluster



## Redis Cluster의 장점/단점

- 장점
  - 자체적인 Primary, Secondary Failover.
  - Slot 단위의 데이터 관리.
- 단점
  - 메모리 사용량이 더 많음
  - Migration 자체는 관리자가 시점을 결정해야 함.
  - Library 구현이 필요함.

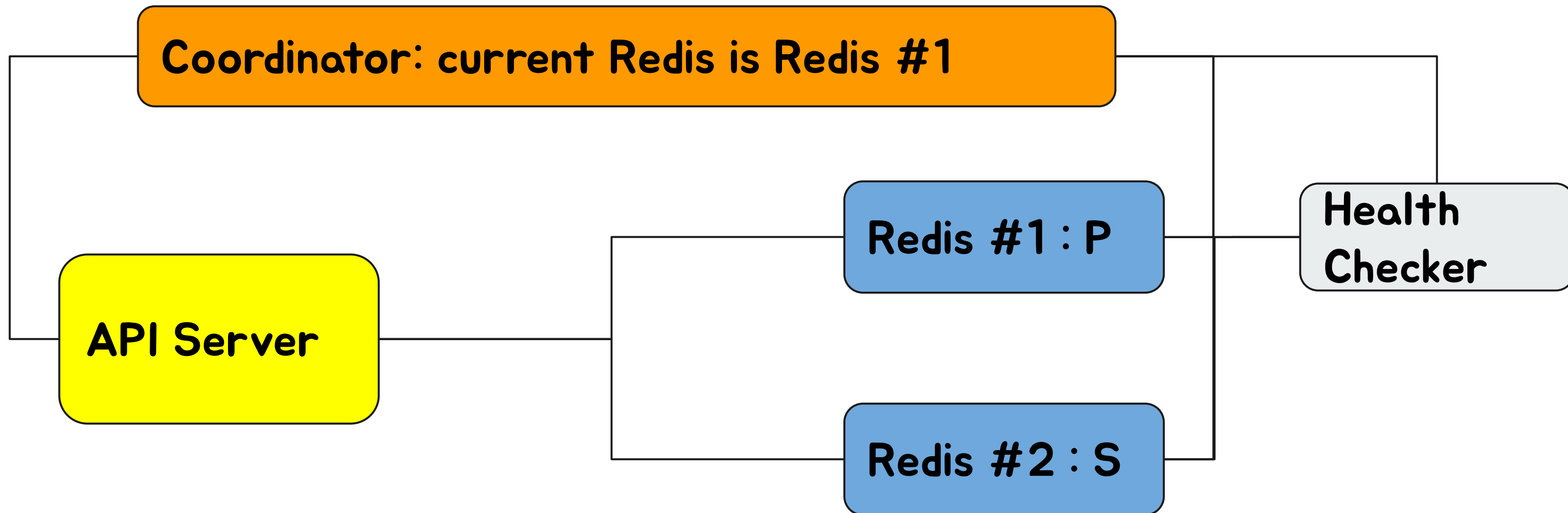
# Redis Failover

## Redis Failover

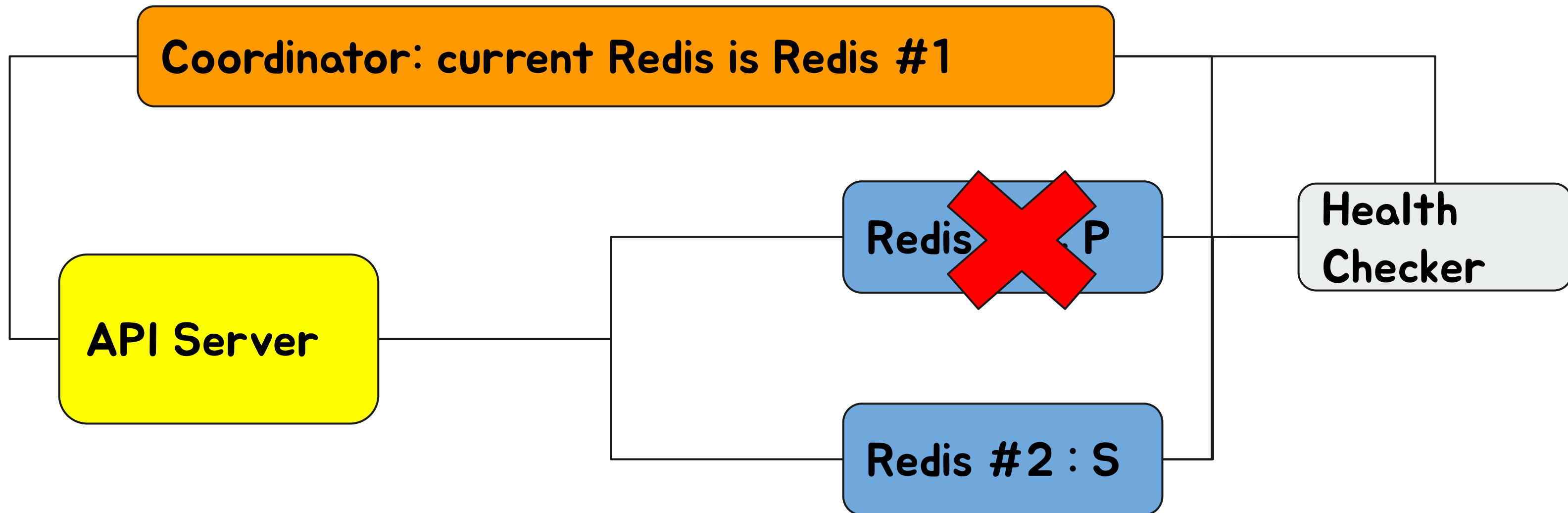
- Coordinator 기반 Failover
- VIP/DNS 기반 Failover
- Redis Cluster 의 사용

# Coordinator 기반

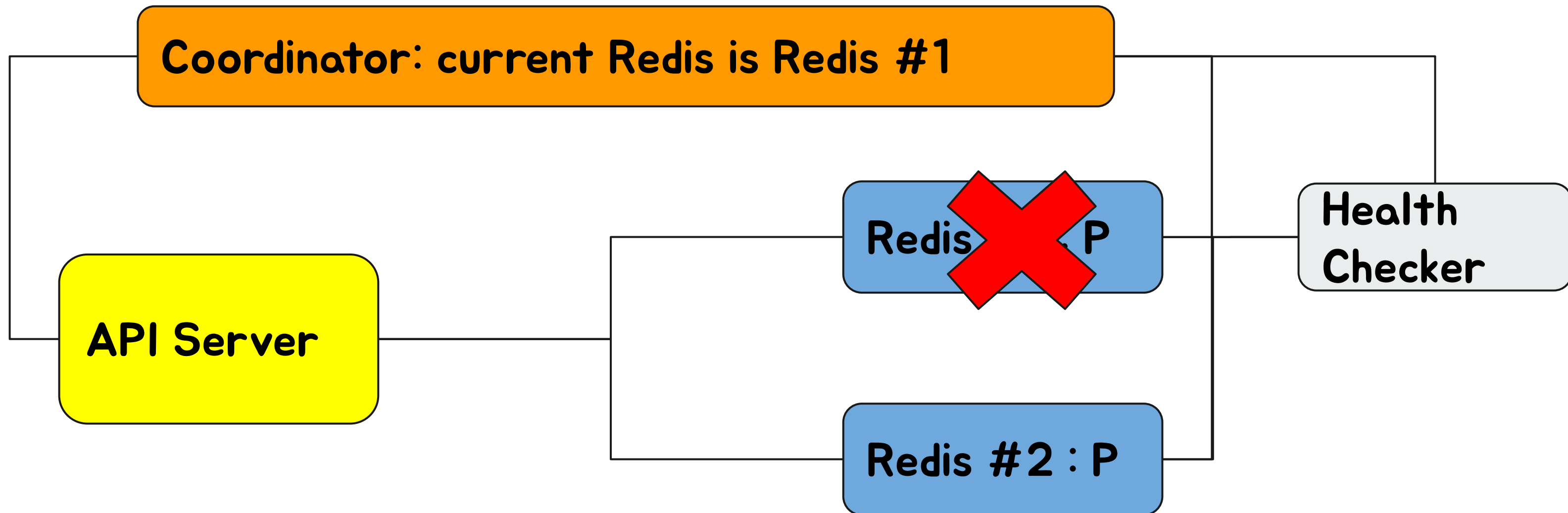
- Zookeeper, etcd, consul 등의 Coordinator 사용



# Coordinator 기반



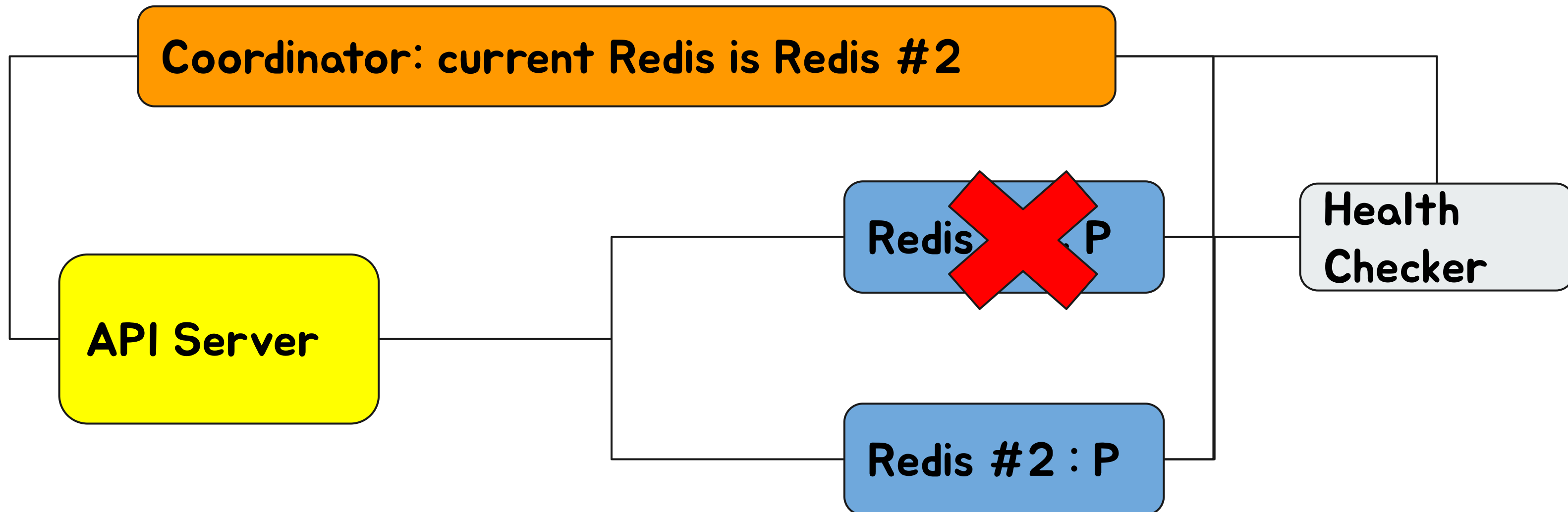
# Coordinator 기반




Health Checker는 Redis #2를 Primary로 승격시킨다.

# Coordinator 기반

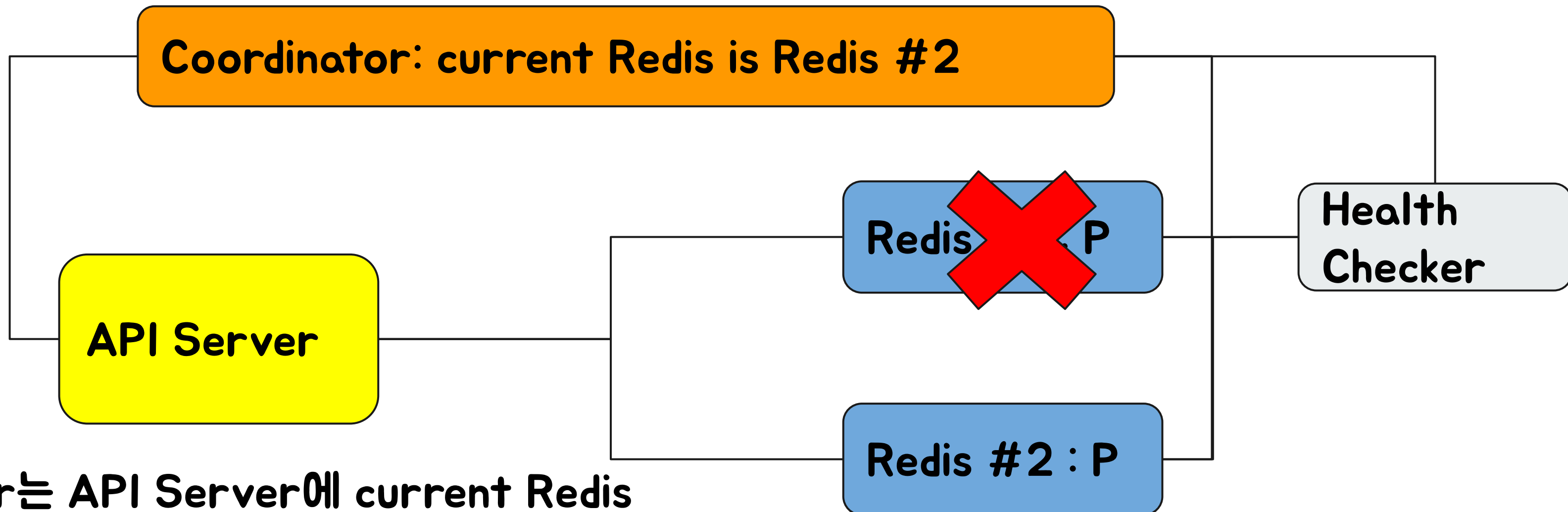
Health Checker는 Coordinator에 current Redis가 Redis #2라고 업데이트 한다.




Health Checker는 Redis #2를 Primary로 승격시킨다.

# Coordinator 기반

Health Checker는 Coordinator에 current Redis가 Redis #2라고 업데이트 한다.




Coordinator는 API Server에 current Redis가 변경되었다고 알려준다.

Health Checker는 Redis #2를 Primary로 승격시킨다.

# Coordinator 기반

- Coordinator 기반으로 설정을 관리한다면 동일한 방식으로 관리가 가능.
- 해당 기능을 이용하도록 개발이 필요하다.

# VIP 기반

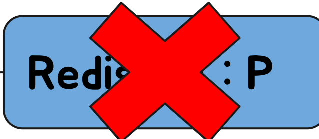
API Server는 10.0.1.1로만 접속

VIP: 10.0.1.1.

# VIP 기반

API Server는 10.0.1.1로만 접속

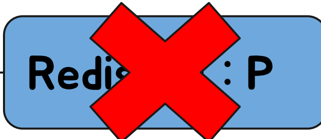
VIP: 10.0.1.1.

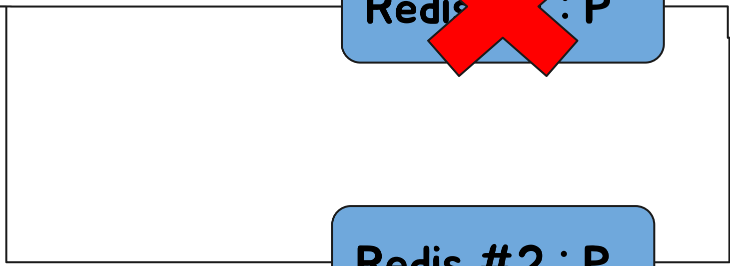


# VIP 기반

API Server는 10.0.1.1로만 접속

VIP: 10.0.1.1.



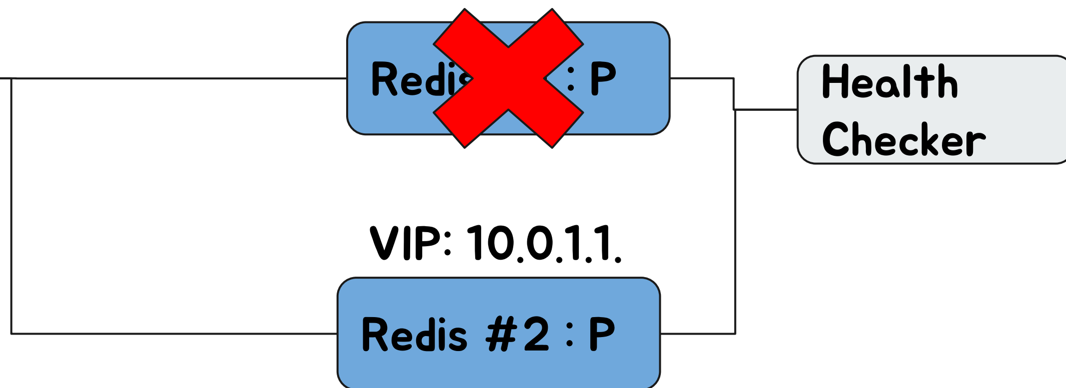



Health Checker는 Redis #2를 Primary
로 승격시킨다.

# VIP 기반

API Server는 10.0.1.1로만 접속

Health Checker는 VIP 10.0.1.1 을 Redis #2
로 할당한다.

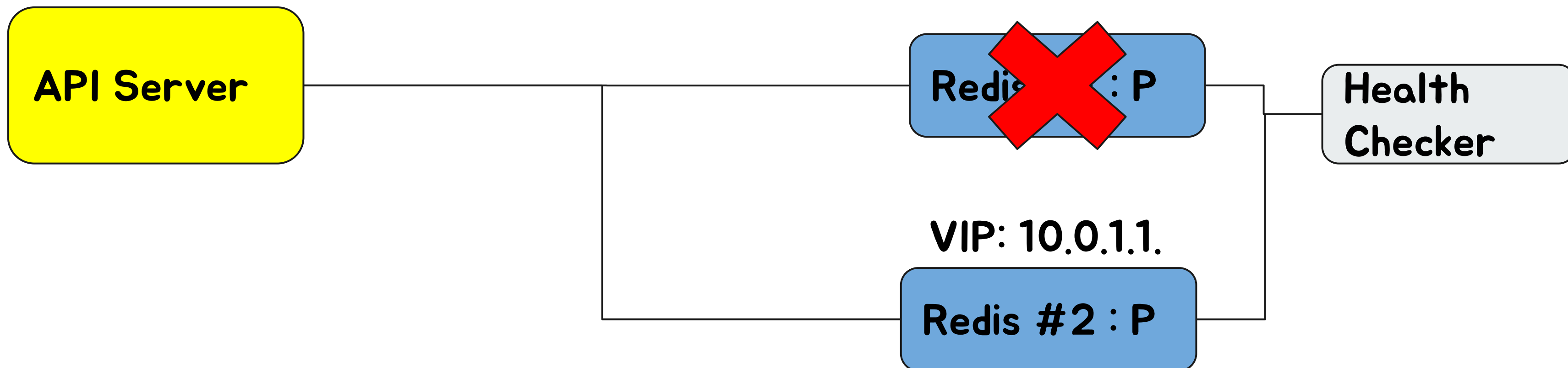



Health Checker는 Redis #2를 Primary
로 승격시킨다.

# VIP 기반

Health Checker는 Redis #1에 있던 기존 연결을 모두 끊어준다.(클라이언트의 재접속 유도)

Health Checker는 VIP 10.0.1.1 을 Redis #2로 할당한다.




Health Checker는 Redis #2를 Primary로 승격시킨다.

# DNS 기반

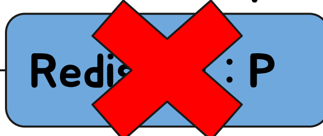
API Server는 redis-primary.svc.io로만 접속

DNS: redis-primary.svc.io

# DNS 기반

API Server는 redis-primary.svc.io로만 접속

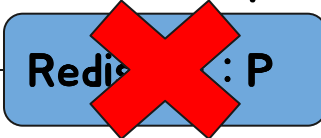
DNS: redis-primary.svc.io

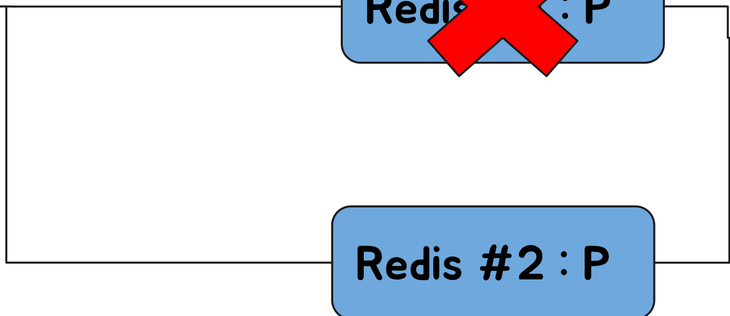


# DNS 기반

API Server는 redis-primary.svc.io로만 접속

DNS: redis-primary.svc.io



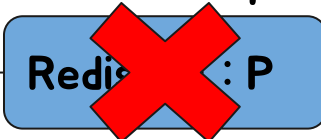


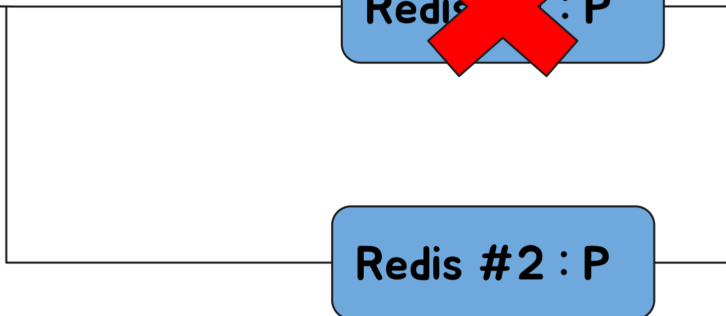

Health Checker는 Redis #2를 Primary
로 승격시킨다.

# DNS 기반

API Server는 redis-primary.svc.io로만 접속

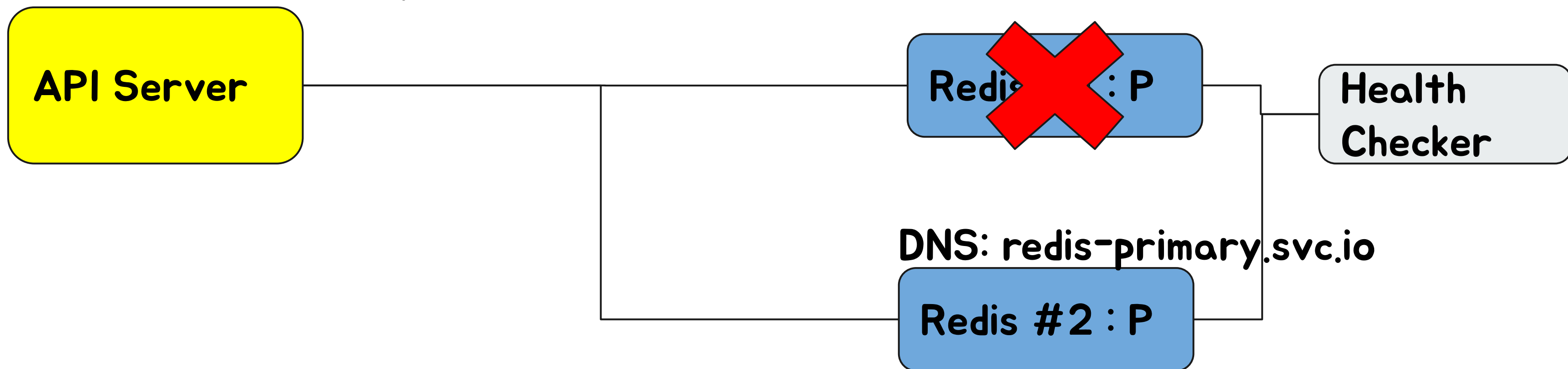
DNS: redis-primary.svc.io






Health Checker는 Redis #2를 Primary
로 승격시킨다.

# DNS 기반

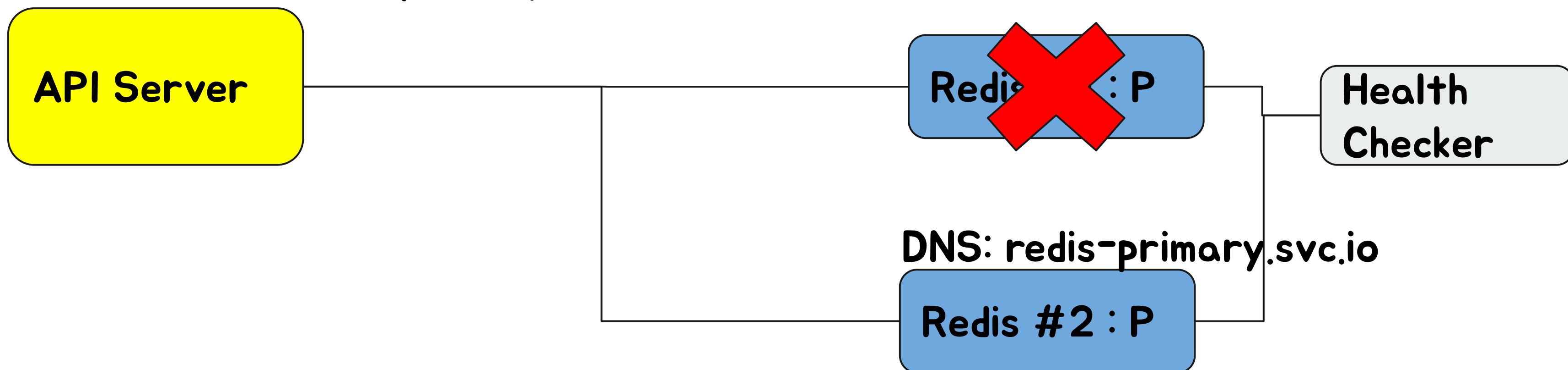


# DNS 기반

Health Checker는 Redis #1에 있던 기존 연결을 모두 끊어준다.(클라이언트의 재접속 유도)

Health Checker는 DNS: redis-primary.svc.io 을 Redis #2 로 할당한다.

API Server는 redis-primary.svc.io로만 접속




Health Checker는 Redis #2를 Primary 로 승격시킨다.

# VIP/DNS 기반

- 클라이언트에 추가적인 구현이 필요없다.
- VIP 기반은 외부로 서비스를 제공해야 하는 서비스 업자에 유리 (예를 들어 클라우드 업체)
- DNS 기반은 DNS Cache TTL을 관리해야 함.
  - 사용하는 언어별 DNS 캐싱 정책을 잘 알아야 함
  - 툴에 따라서 한번 가져온 DNS 정보를 다시 호출 하지 않는 경우도 존재

# Monitoring

# Monitoring Factor

- Redis Info를 통한 정보
  - RSS
  - Used Memory
  - Connection 수
  - 초당 처리 요청 수
- System
  - CPU
  - Disk
  - Network rx/tx

# CPU가 100%를 칠 경우

- 처리량이 매우 많다면?
  - 좀 더 CPU 성능이 좋은 서버로 이전
  - 실제 CPU 성능에 영향을 받음
    - 그러나 단순 get/set은 초당 10만 이상 처리가능
- O(N) 계열의 특정 명령이 많은 경우.
  - Monitor 명령을 통해 특정 패턴을 파악하는 것이 필요
  - Monitor 잘못쓰면 부하로 해당 서버에 더 큰 문제를 일으킬 수도 있음.(짧게 쓰는게 좋음)

# 결론

## 결론

- 기본적으로 Redis는 매우 좋은 툴
- 그러나 메모리를 빡빡하게 쓸 경우, 관리하기가 어려움
  - 32기가 장비라면 24기가 이상 사용하면 장비 증설을 고려하는 것이 좋음.
  - Write가 Heavy 할 때는 migration도 매우 주의해야함.
- Client-output-buffer-limit 설정이 필요.

# Redis as Cache

- Cache 일 경우는 문제가 적게 발생
  - Redis 가 문제가 있을 때 DB등의 부하가 어느정도 증가하는 지 확인 필요.
  - Consistent Hashing도 실제 부하를 아주 균등하게 나누지는 않음. Adaptive Consistent Hashing 을 이용해 볼 수도 있음.

# Redis as Persistent Store

- Persistent Store의 경우
  - 무조건 Primary/Secondary 구조로 구성이 필요함
  - 메모리를 절대로 빡빡하게 사용하면 안됨.
    - 정기적인 migration이 필요.
    - 가능하면 자동화 툴 을 만들어서 이용
  - RDB/AOF가 필요하다면 Secondary에서만 구동

# 감사합니다.