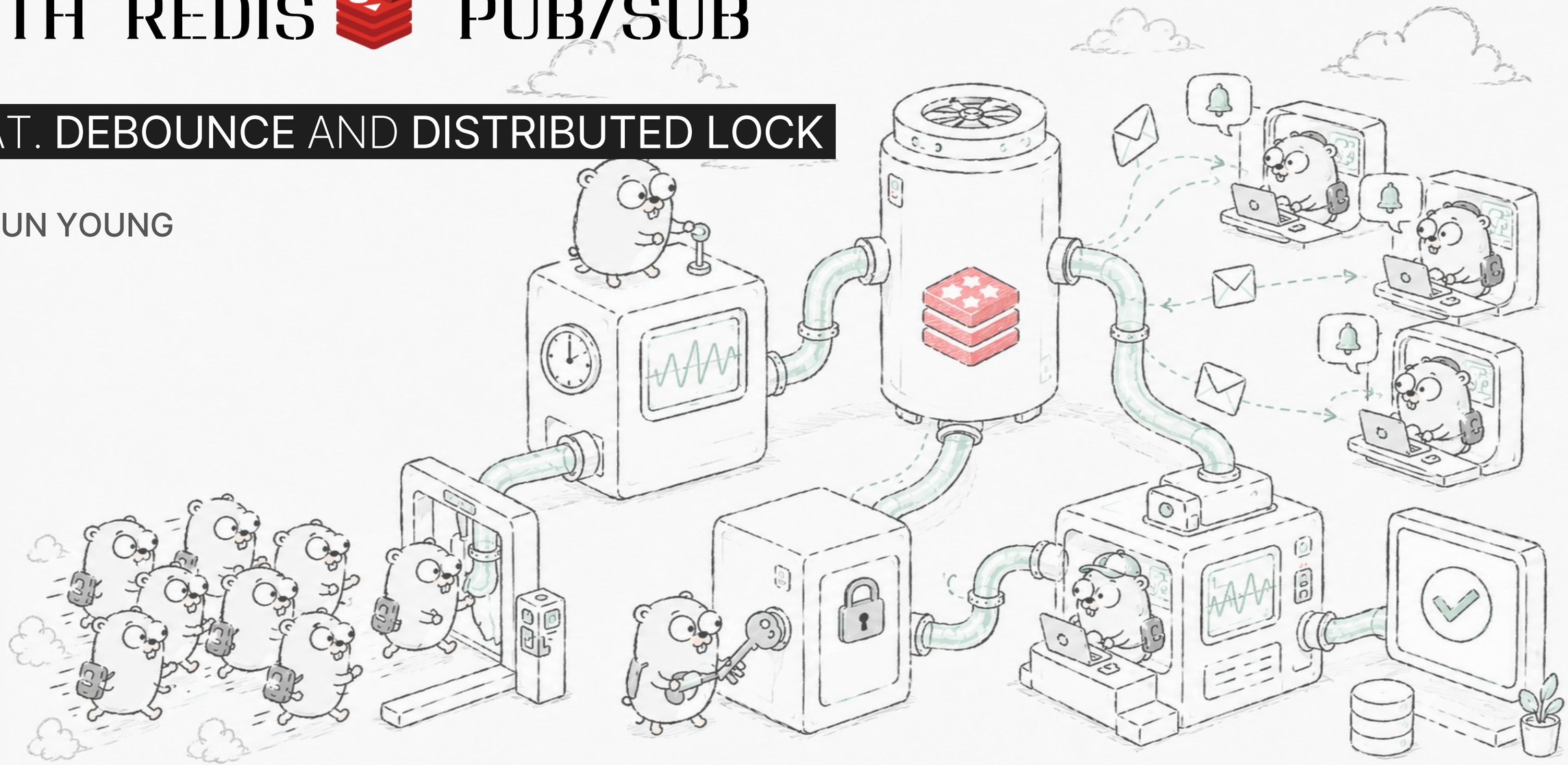
MITIGATING THUNDERING HERD WITH REDIS PUB/SUB

FEAT. DEBOUNCE AND DISTRIBUTED LOCK

KIM JUN YOUNG



The above cover was created using AI.

Introduction

Kim JunYoung *김준영*

現 세명컴퓨터고등학교 스마트보안솔루션과 3학년 (11기)

現 세명컴퓨터고등학교 스마트보안솔루션과 대표 (학생회)

現 교내 개발/클라우드 컴퓨팅/DevOps 전공동아리 Null4U 부장

Github	github.com/**yulmwu**
Discord	@rlawnsdud
Phone	010-2980-1336
Email	• me@swua.kr • normal8781@gmail.com
Linkedin	linkedin.com/in/yulmwu

Homepage

https://swua.kr

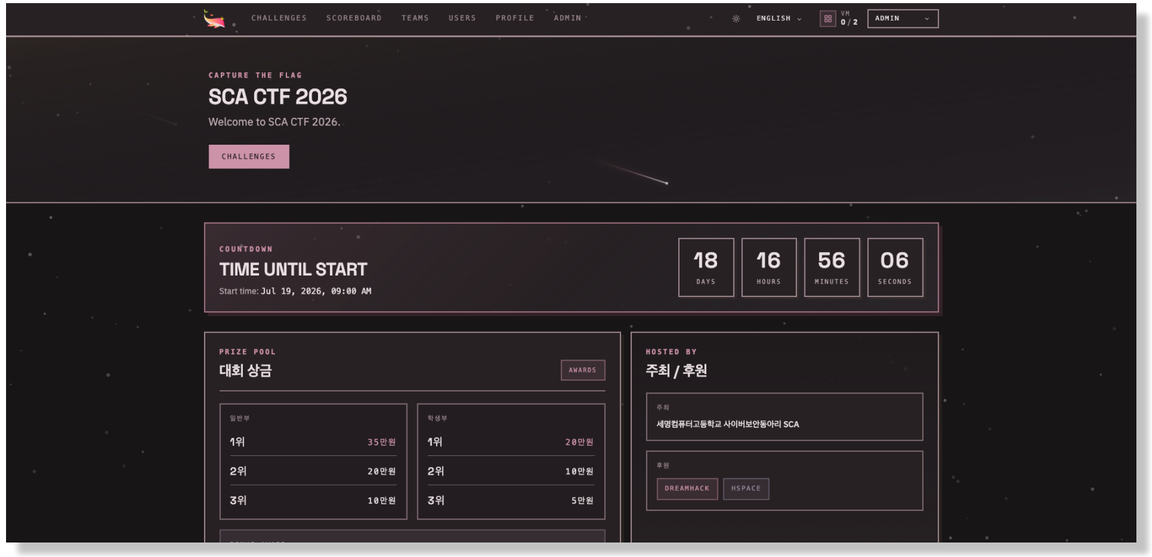
Article (Blog)

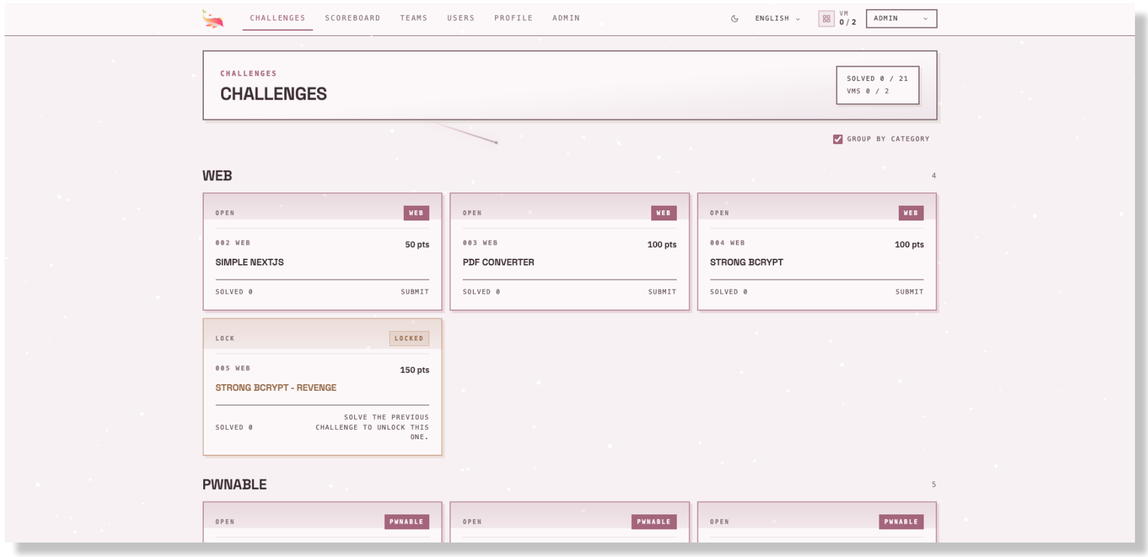
Mitigating Thundering Herd with Redis Pub/Sub
(feat. Debounce and Distributed Lock), 2026.02.27

Table of Contents

서론 및 개요

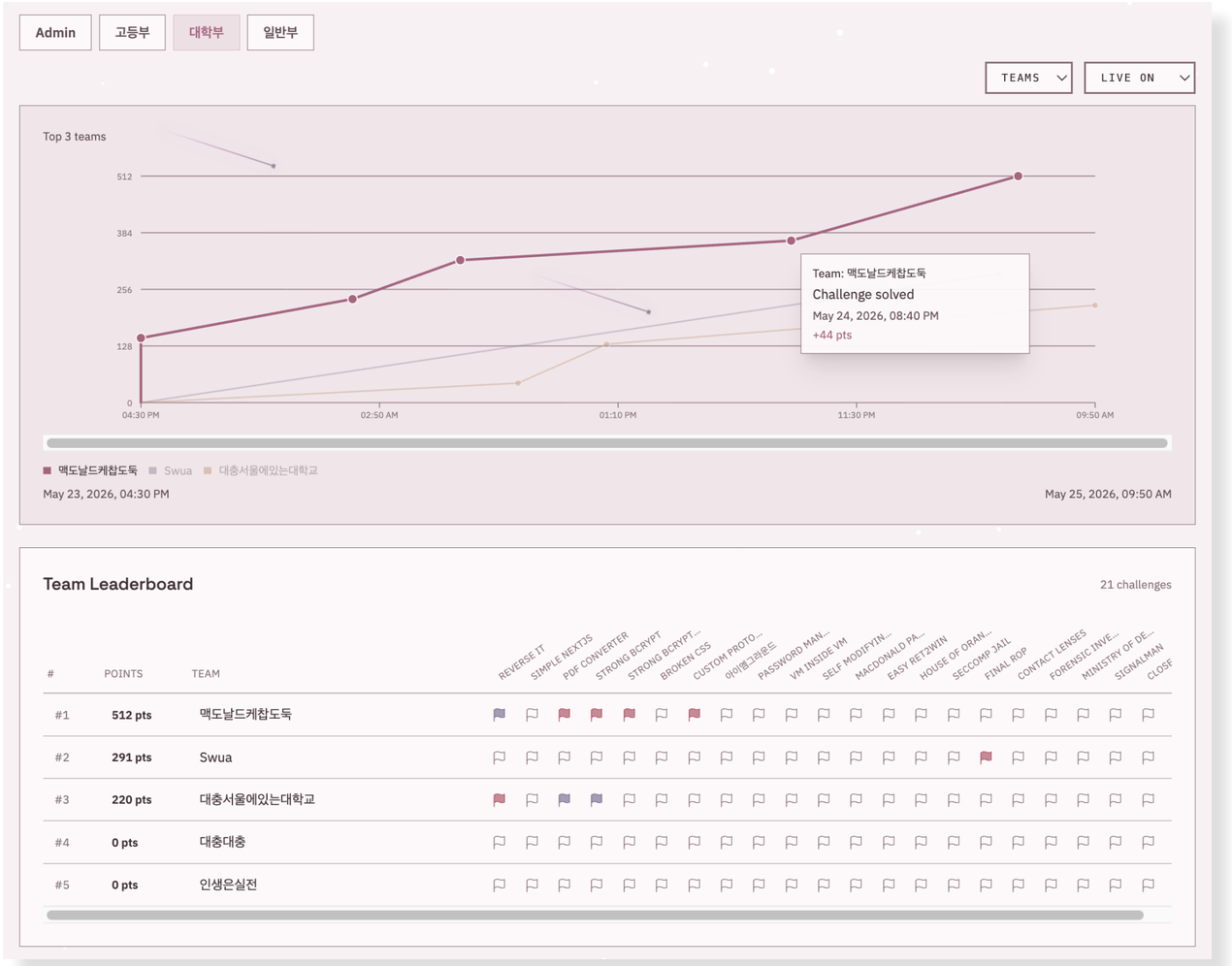
2026년 초, SCA CTF 운영을 위해 CTF 플랫폼을 자체적으로 개발하기로 하였음. (SMCTF)





이를 바탕으로 2026년 SCA CTF를 운영할 예정이고, 자체적으로 개발된 Sandboxd-O Sandbox-Like 오케스트레이션 플랫폼을 통해 VM 환경을 제공할 수 있도록 정성있게 개발함.

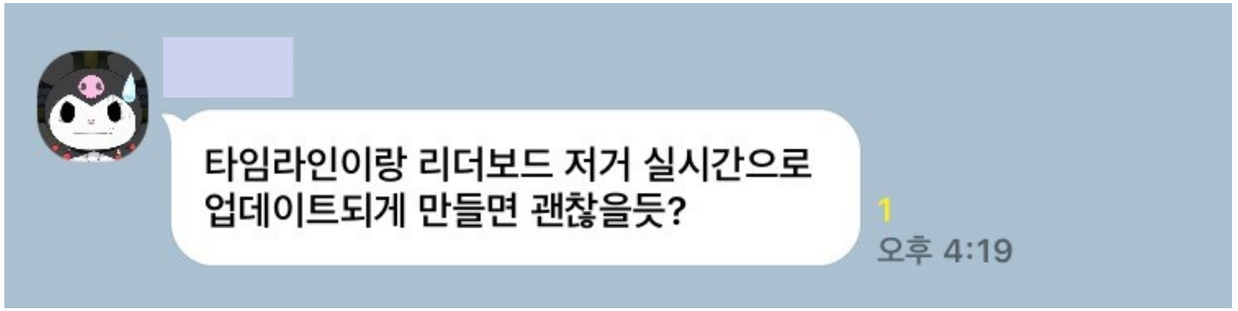
서론 및 개요



CTF 플랫폼에서 가장 중요한 기능 중 하나가 **스코어보드**이지 않을까 생각함.

SMCTF에서도 타임라인과 리더보드 2가지 형태의 스코어보드를 지원하는데, 전자의 경우 시간대별로 풀이를 확인, 후자는 순위와 어떤 문제를 풀었는지, 또 First Blood 인지를 나타냄.

서론 및 개요

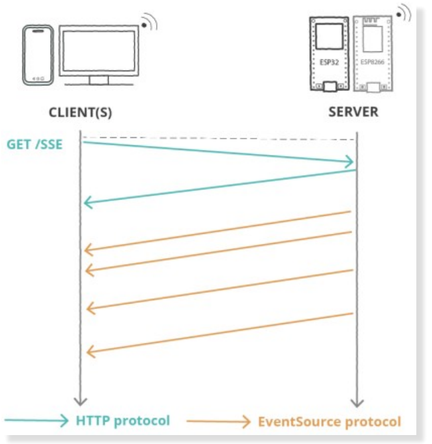


" **실시간 업데이트 기능** 구현해줘 "

어떤 기술로?

Pulling or Polling

WebSocket



SSE(Server-Sent Events)

서론 및 개요

Pulling or Polling

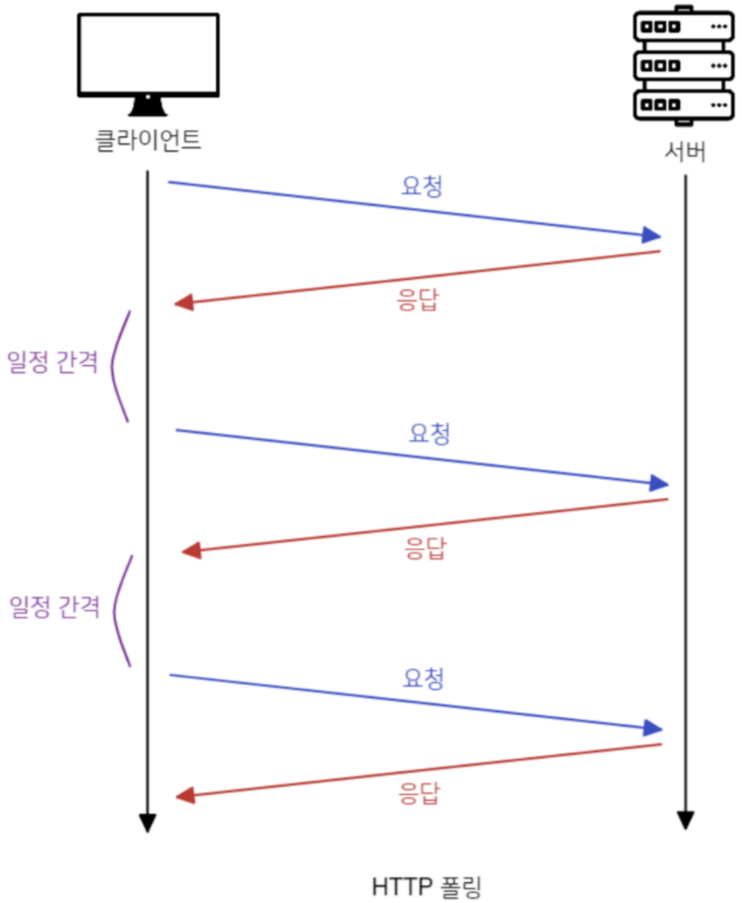
주기적으로 서버에 요청(Pull)을 보내 상태를 동기화하는 것. (Polling)

장점

- 구현이 매우 간단함.

단점

- 서버에 대한 부담이 매우 커짐
- 불필요한 리소스 및 트래픽이 낭비됨
- 특정한 간격이 있기 때문에 사실 실시간이라고 볼 수 없음.



HTTP 폴링

(일반적인 Polling의 단점을 보완한 Long Polling도 있으나, 이것도 단점이 같음)

서론 및 개요

WebSocket

TCP Connection을 통해 양방향으로 통신을 이어지게 하는 표준 기술.

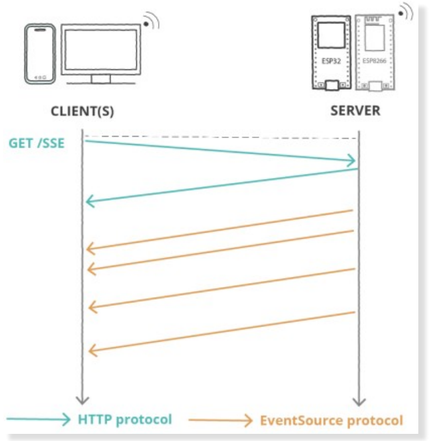
장점

- 실시간 양방향 통신임
- 초기 Connection(HandShake)을 제외하면 오버헤드가 거의 없음
- 한번의 Connection으로 지속적으로 통신 가능

단점

- 서버 리소스 소모가 좀 있음
- 구현이 복잡함 (특히 분산 환경. Discord의 Gateway를 생각하면 됨)

서론 및 개요



SSE
(Server-Sent Events)

서버에서 클라이언트, 즉 단방향으로 데이터를 전송할 수 있는 기술.

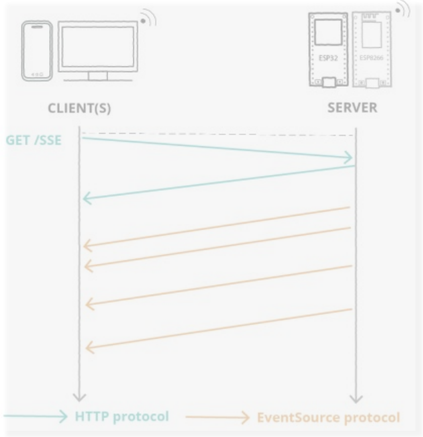
장점

- 순수 HTTP 프로토콜 위에서 동작하기 때문에 구현이 매우 쉬움
- 가벼움

단점

- 서버에서 클라이언트로의 단방향 통신만 됨 (단, 의도했던 요소)
- **WebSocket과 마찬가지로 분산된 환경에서 적절한 비즈니스 로직이 있어야함 (*중요)**

서론 및 개요



SSE
(Server-Sent Events)

서버에서 클라이언트, 즉 단방향으로 데이터를 전송할 수 있는 기술.

장점

- 순수 HTTP 프로토콜 위에서 동작하기 때문에 구현이 매우 쉬움
- 가벼움

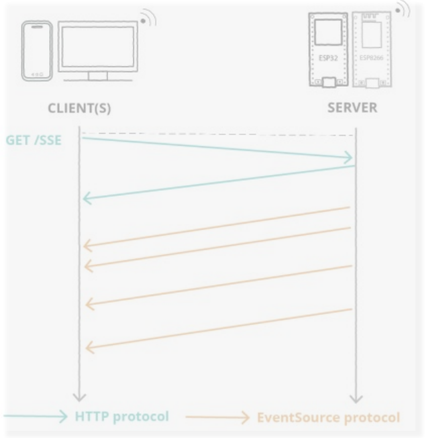
단점

- 서버에서 클라이언트로의 단방향 통신만 됨 (단, 의도했던 요소)
- WebSocket과 마찬가지로 분산된 환경에서 적절한 비즈니스 로직이 있어야함 (*중요)

Polling 방식은 매우 비효율적이기 때문에 사용하지 않고, WebSocket은 오버엔지니어링이라 판단함.

요구사항을 만족하는 가장 좋은 솔루션은 **SSE**(Server-Sent Events).

서론 및 개요



SSE
(Server-Sent Events)

서버에서 클라이언트, 즉 단방향으로 데이터를 전송할 수 있는 기술.

장점

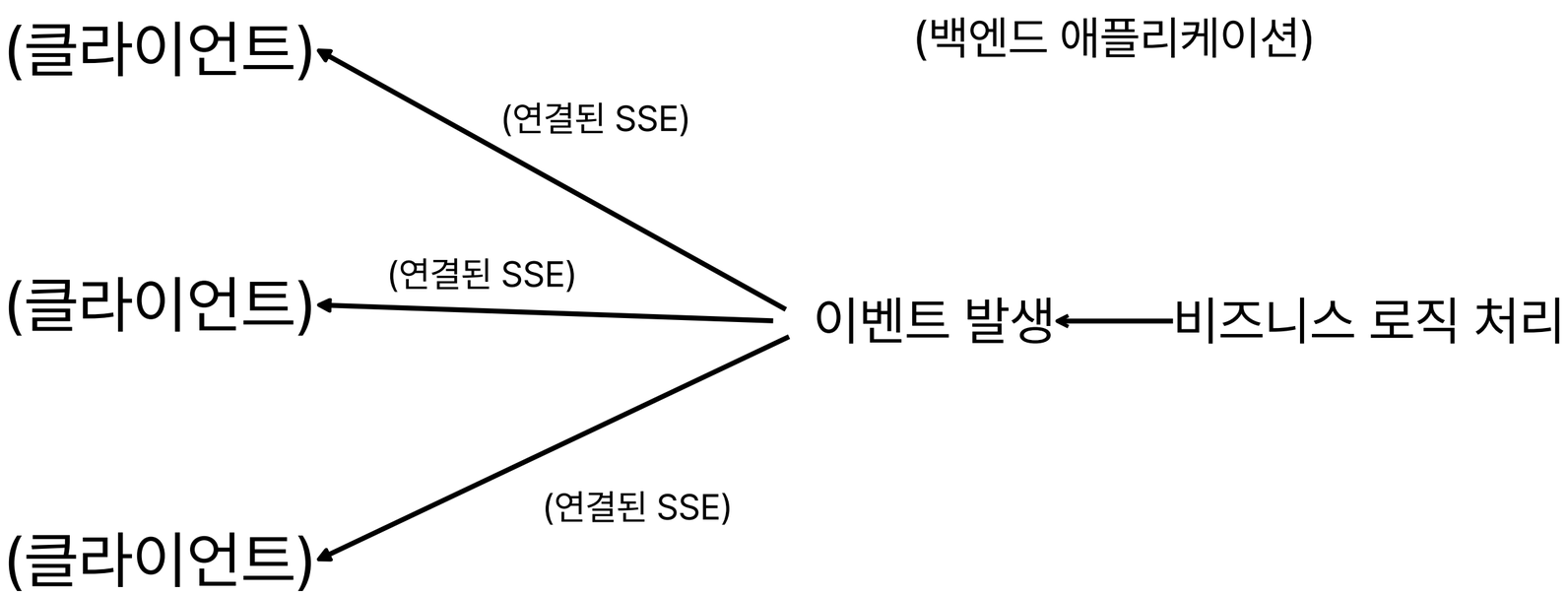
- 순수 HTTP 프로토콜 위에서 동작하기 때문에 구현이 매우 쉬움
- 가벼움

단점

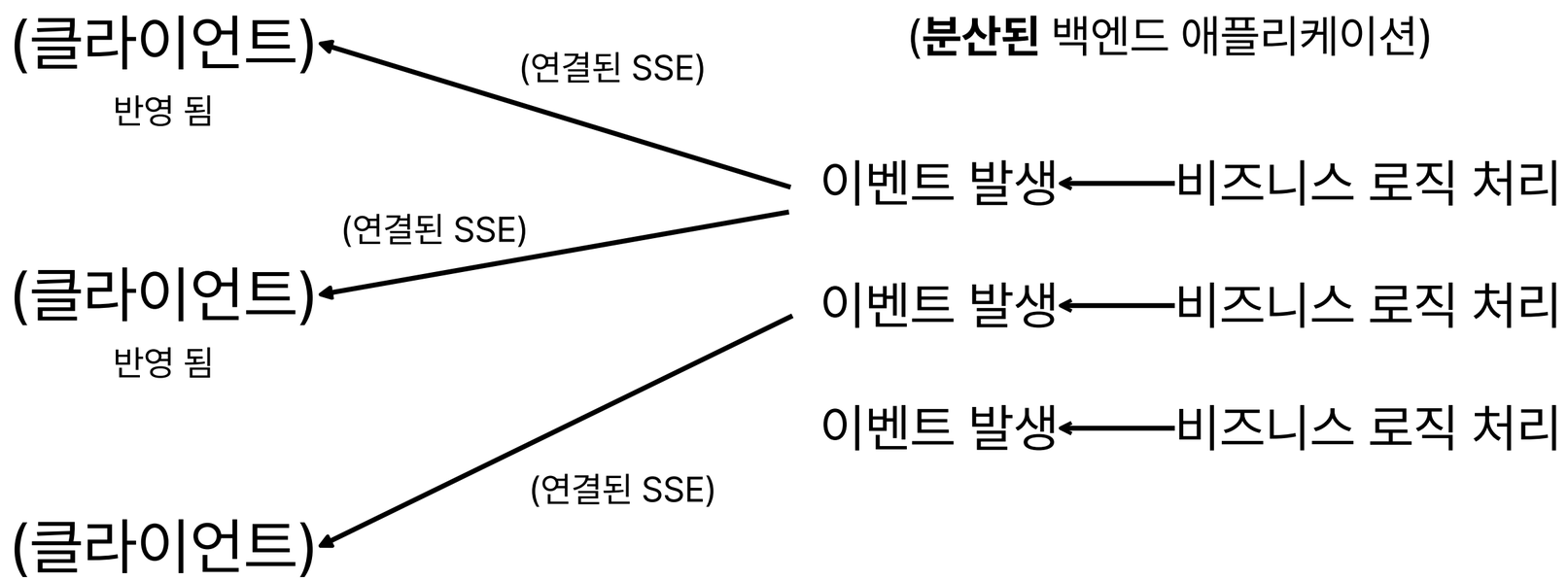
- 서버에서 클라이언트로의 단방향 통신만 됨 (단, 의도했던 요소)
- WebSocket과 마찬가지로 분산된 환경에서 적절한 비즈니스 로직이 있어야함 (*중요)

`if (` 올바른 플래그가 제출되었는가?
문제가 업데이트되었는가?
유저나 팀이 추가되었는가? `) then` 클라이언트와 연결된
SSE에 업데이트된 스코어보드
데이터 이벤트 전송

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



다른 애플리케이션에서 처리되어 SSE 이벤트 트리거가 안됨 (반영 안됨)

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

```go
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()      스코어보드 캐싱 무효화
        h.invalidateLeaderboardCache()  SSE 이벤트 전송
        h.publishScoreboardEvent() // 예시
        // ... (중략)
    }     A 애플리케이션에서 위 로직이 실행되면 B 애플리케이션에서

          SSE로 연결된 클라이언트는 이벤트를 받을 수 없음. (SSE 이벤트 자체가 해당 애플리케이션에서 작동을 안함)
    // ... (중략)
}
```

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

그 전에.. **스코어보드 캐싱**

```
30  > func (r *ScoreboardRepo) leaderboardChallenges(ctx context.Context, divisionID *int64) ([]models.LeaderboardChallenge, map[int64]int, error)
69    }
70
71  > func (r *ScoreboardRepo) Leaderboard(ctx context.Context, divisionID *int64) (models.LeaderboardResponse, error) { ...
180   }
181
182 > func (r *ScoreboardRepo) TeamLeaderboard(ctx context.Context, divisionID *int64) (models.TeamLeaderboardResponse, error) { ...
305   }
306
307 > func (r *ScoreboardRepo) TimelineSubmissions(ctx context.Context, since *time.Time, divisionID *int64) ([]models.UserTimelineRow, error) {
340   }
341
342 v func (r *ScoreboardRepo) TimelineTeamSubmissions(ctx context.Context, since *time.Time, divisionID *int64) ([]models.TeamTimelineRow, error)
```

Line 30 to 400+

스코어보드 계산은 내부적으로 복잡한 쿼리로 인해 무거운 작업임. 그래서 매 요청마다 계산하면 매우 큰 오버헤드 발생.

매번 새롭게 계산하는게 아닌, **스코어보드가 변경될때**만 다시 계산하고 어딘가 저장해두기(캐싱해두기)

```
if (  올바른 플래그가 제출되었는가?
      문제가 업데이트되었는가?
      유저나 팀이 추가되었는가?  )
```

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

그 전에.. **스코어보드 캐싱**

```go
func (h *Handler) Leaderboard(ctx *gin.Context) {
    cacheKey := "leaderboard:users"
    if h.respondFromCache(ctx, cacheKey) {
        return
    }

    rows, err := h.score.Leaderboard(ctx.Request.Context())
    if err != nil {
        writeError(ctx, err)
        return
    }

    h.storeCache(ctx, cacheKey, rows, h.cfg.Cache.LeaderboardTTL)
    ctx.JSON(http.StatusOK, rows)
}
```

스코어보드(타임라인, 리더보드) API 호출 시

```go
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()
        h.invalidateLeaderboardCache()
        // ... (중략)
    }

    // ... (중략)
}
```

Redis 캐싱이 되어있다면(Cache Hit) 바로 응답

Cache Miss 라면 Leaderboard 계산 및 캐싱 후 응답

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

그 전에.. **스코어보드 캐싱**

```go
func (h *Handler) Leaderboard(ctx *gin.Context) {
    cacheKey := "leaderboard:users"
    if h.respondFromCache(ctx, cacheKey) {
        return
    }

    rows, err := h.score.Leaderboard(ct...ext())
    if err != nil {
        writeError(ctx, err)
        return
    }

    h.storeCache(ctx, cacheKey, rows, h.cfg.Cache...
    ctx.JSON(http.StatusOK,...
}
```

스코어보드(타임라인, 리더보드) API 호출 시

Redis 캐싱이 되어있다면(Cache Hit) 바로 응답

Cache Miss 라면 Leaderboard 계산 및 캐싱 후 응답

```go
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()
        h.invalidateLeaderboardCache()
        // ... (중략)
    }

    ... (중략)
}
```

redis

고성능 인메모리 DB (영속성 지원)

원자성 보장

매우 유연함 등등 ...

캐싱이나 세션 관리, 메시지 큐 등에서 매우 유용하게 쓰임

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

```
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()        스코어보드 캐싱 무효화
        h.invalidateLeaderboardCache()     SSE 이벤트 전송
        h.publishScoreboardEvent() // 예시
        // ... (중략)
    }
                스코어보드 재계산을 위해 캐싱 무효화 후 SSE 이벤트 전송.
                하지만 분산된 환경에선 여러 애플리케이션간 일관적으로 SSE로 이벤트 전송이 불가능한데..
    // ... (중략)
}
```

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

```
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()
        h.invalidateLeaderboardCache()
        h.publishScoreboardEvent() // 예시
        // ... (중략)
    }

    // ... (중략)
}
```

스코어보드 캐싱 무효화

SSE 이벤트 전송

스코어보드 재계산을 위해 캐싱 무효화 후 SSE 이벤트 전송.

하지만 분산된 환경에선 여러 애플리케이션간 일관적으로 SSE로 이벤트 전송이 불가능한데..

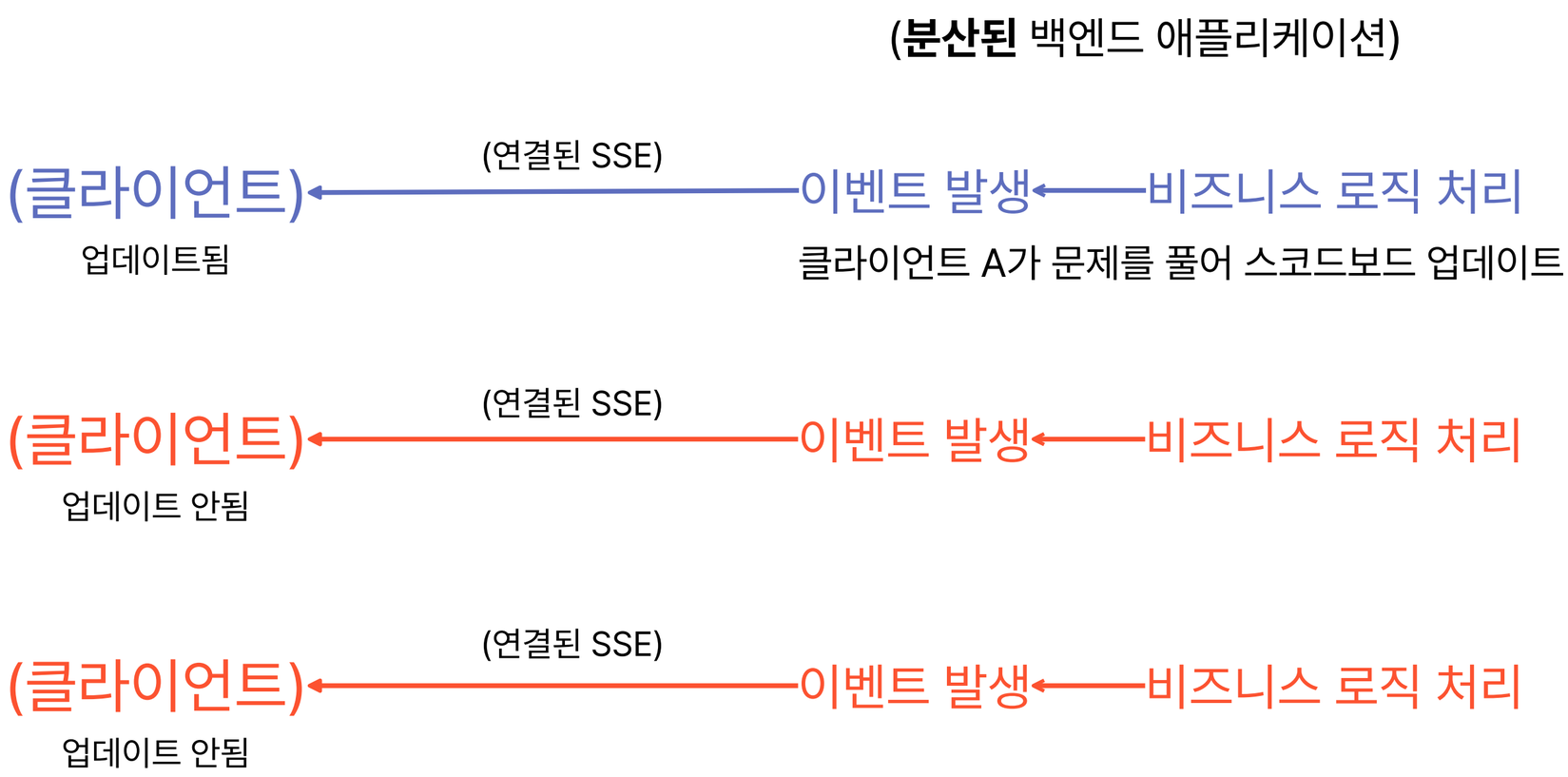
그럼 중간에서 허브 같은걸 두면 되지 않을까?

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

(**분산된** 백엔드 애플리케이션)

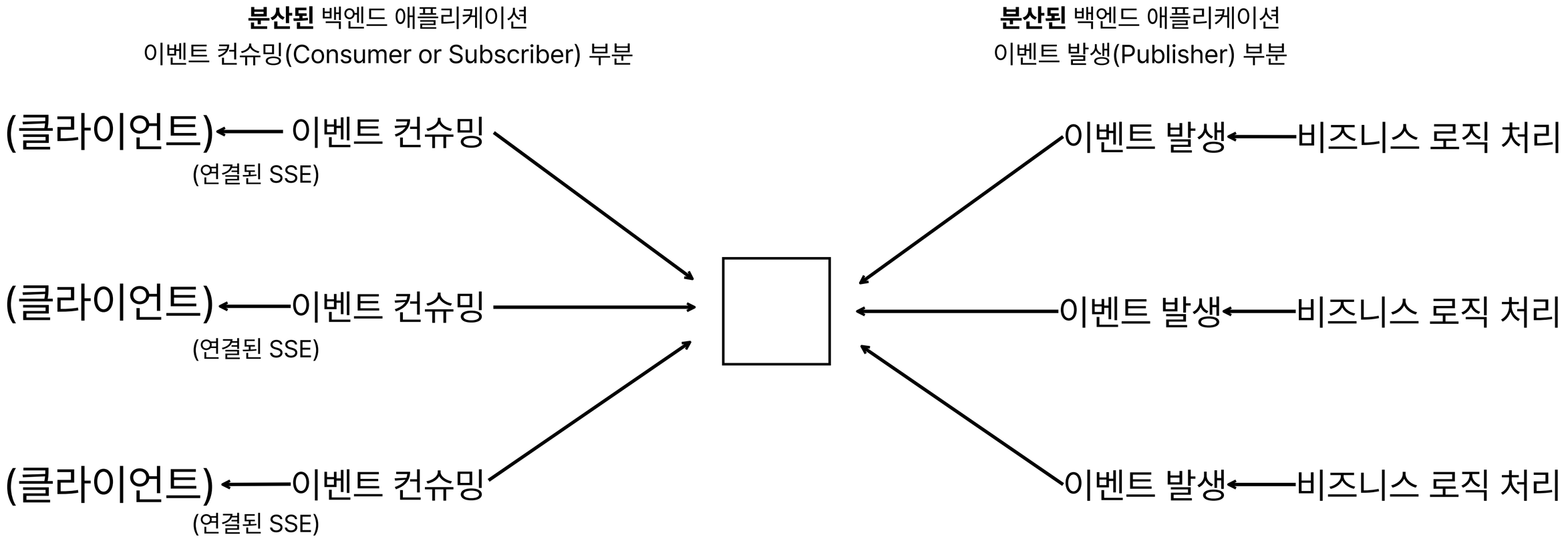
각 백엔드 애플리케이션이 독립적으로 스코어보드 연산을 처리하고, 자기 자신에게 연결된 SSE 커넥션으로만 요청을 보내는 문제

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



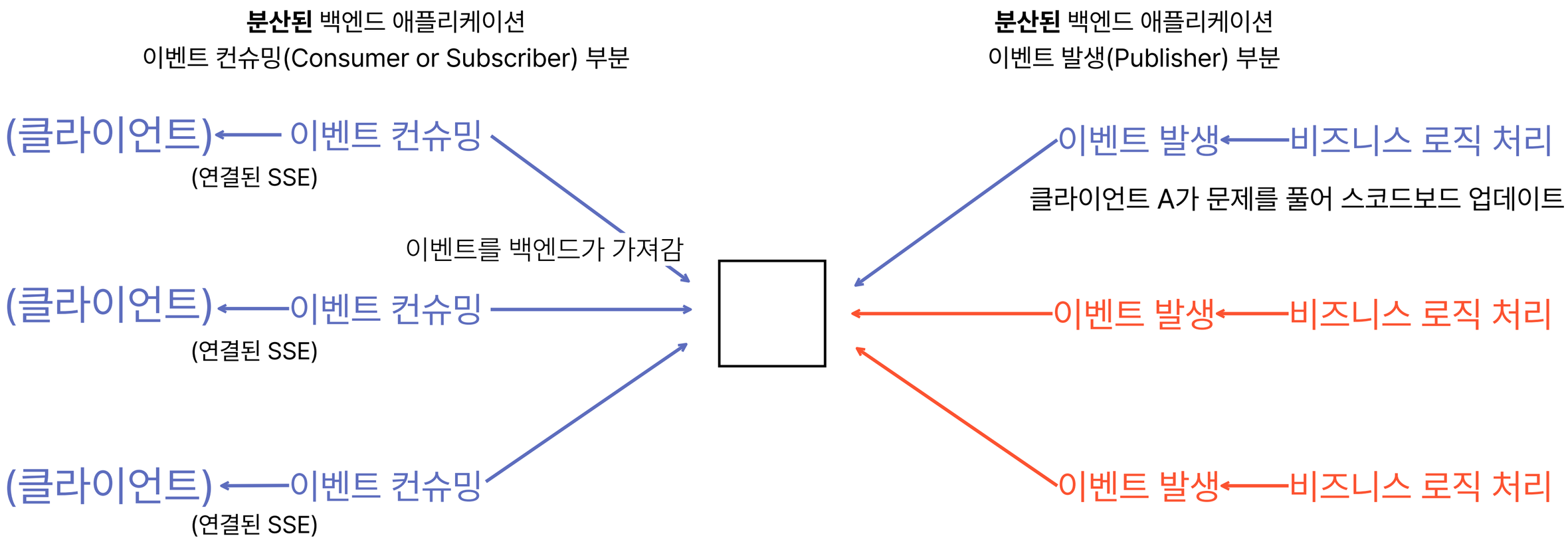
각 백엔드 애플리케이션이 독립적으로 스코어보드 연산을 처리하고, 자기 자신에게 연결된 SSE 커넥션으로만 요청을 보내는 문제

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



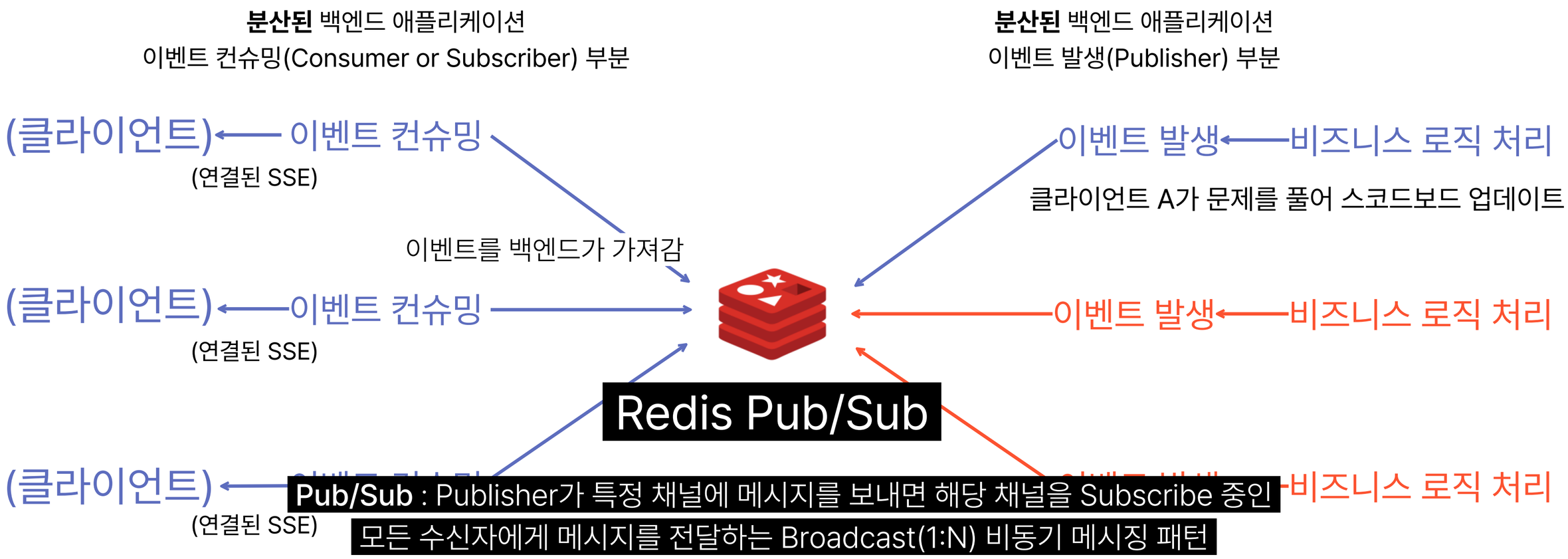
백엔드 애플리케이션에서 이벤트(메시지) 발생시 하나의 허브를 둬서 이벤트를 공유하는 것. (**Pub/Sub** 메시징 시스템)

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



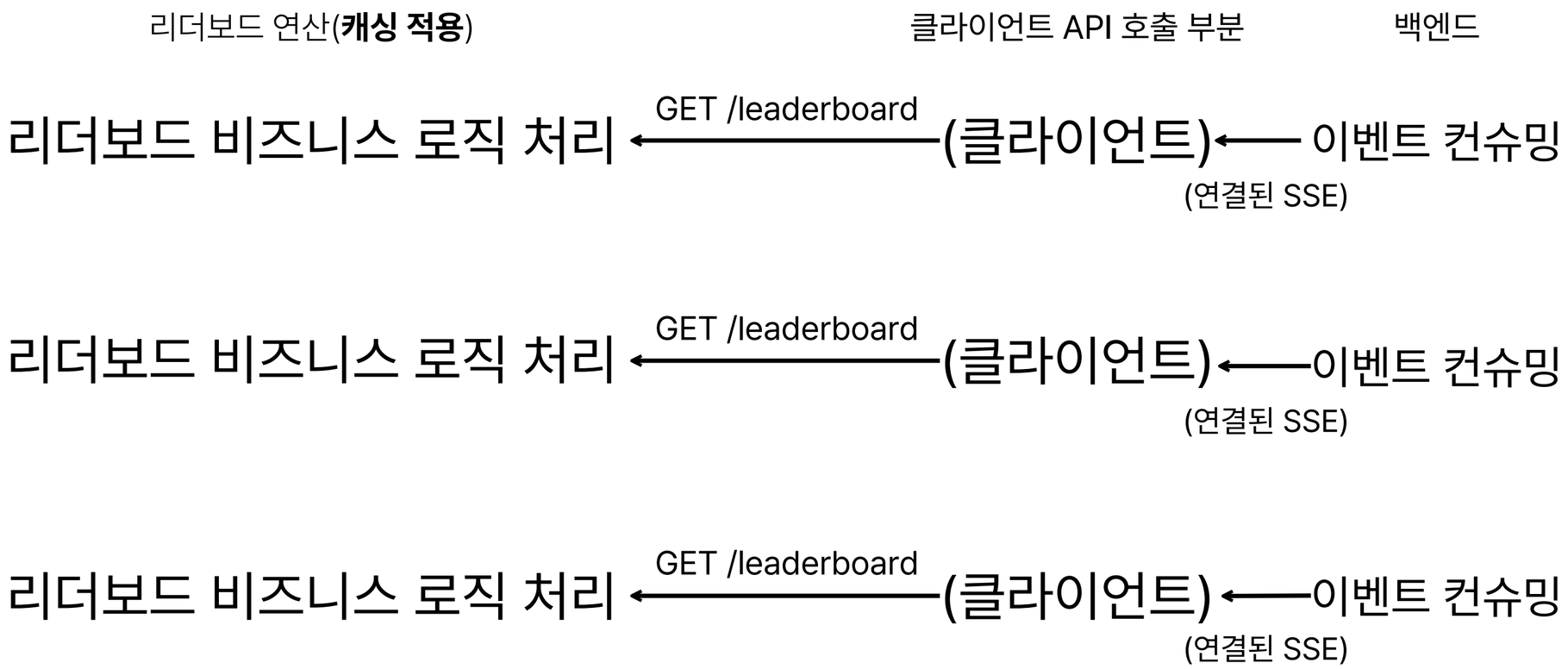
백엔드 애플리케이션에서 이벤트(메시지) 발생시 하나의 허브를 둬서 이벤트를 공유하는 것. (**Pub/Sub** 메시징 시스템)

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?



백엔드 애플리케이션에서 이벤트(메시지) 발생시 하나의 허브를 둬서 이벤트를 공유하는 것. (**Pub/Sub** 메시징 시스템)

Redis Pub/Sub 도입하기 — 왜 분산 환경에서 **메시징 시스템**이 필요한가요?

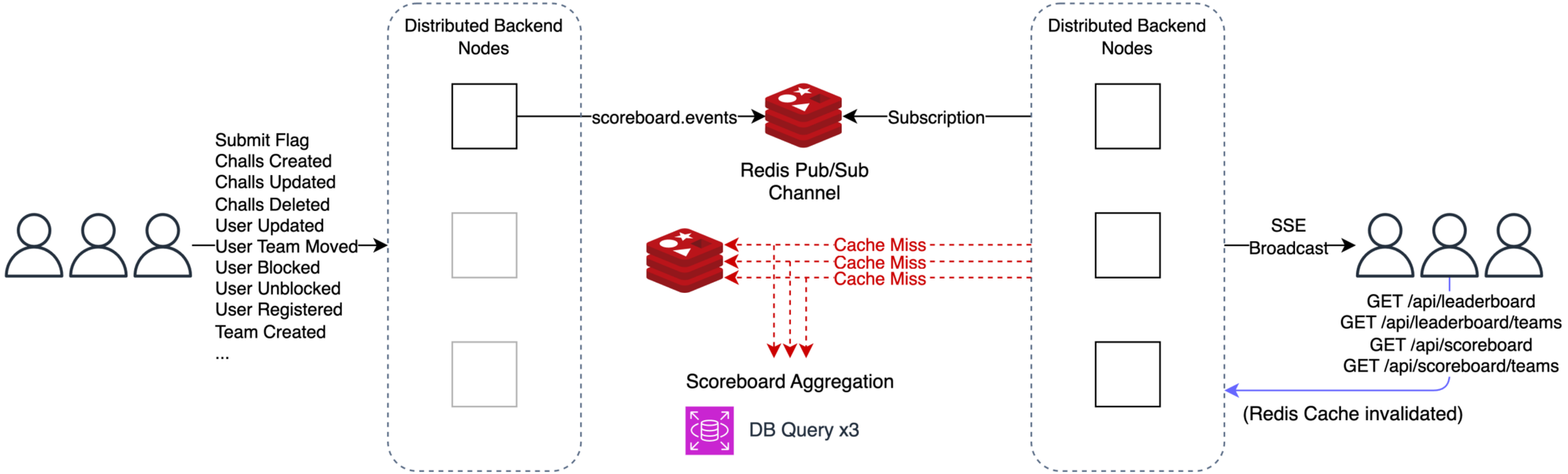


이후 '스코어보드가 변경되었다' 라는 이벤트(메시지)를 SSE를 통해 받은 클라이언트가 스코어보드 관련 API 호출

(SSE를 통해 직접 페이로드를 전송하지 않는 이유는 효율성 및 향후 확장성, 그리고 유지보수 차원에서 코드 변경을 최소화하기 위함)

Thundering Herd 문제 해결하기

그런데 SSE로 받는 이벤트는 단순히 알림이기 때문에, 클라이언트에서 새롭게 스코어보드 API를 하는 구조이고,
이는 곧 **거의 같은 시간에 여러 요청이 Burst로 들어오고** 다수의 **Cache Miss***가 발생하여 **Thundering Herd** 문제가 발생할 수 있음.

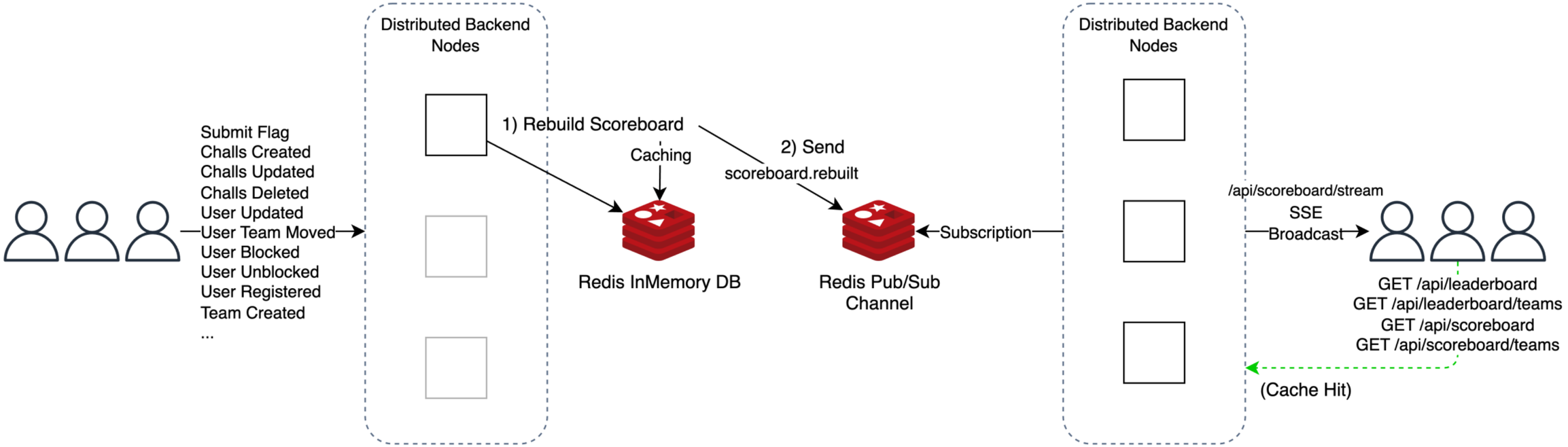


(아키텍처에서 Distributed Backend Nodes와 Redis 모양이 여러개지만, 이는 이해를 돕기 위함임)

※ 스코어보드 연산 속도(100ms 이상)를 무시할 수 없기 때문에 대규모 환경이나 앞선 Burst 요청에 대해 Cache Miss가 충분히 발생 가능함.

Thundering Herd 문제 해결하기

이러한 문제를 해결하기 위해 처음 고안한 구조는 다음과 같음.

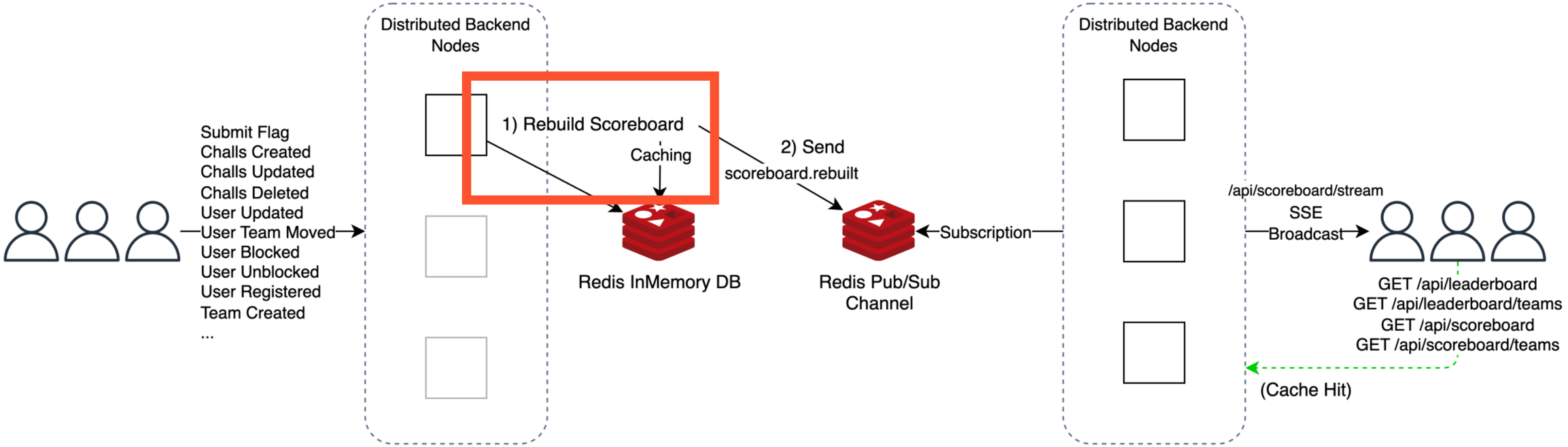


사용자가 스코어보드에 영향을 미치는 API를 호출한다면, 분산된 환경 중 해당 백엔드 노드에서 스코어보드를 집계하고 계산함.

이때 캐싱을 무효화하는 것은 마찬가지이나, 이 작업이 완료된 뒤에 Pub/Sub 메시징 채널이 이벤트를 보내는 것.

Thundering Herd 문제 해결하기

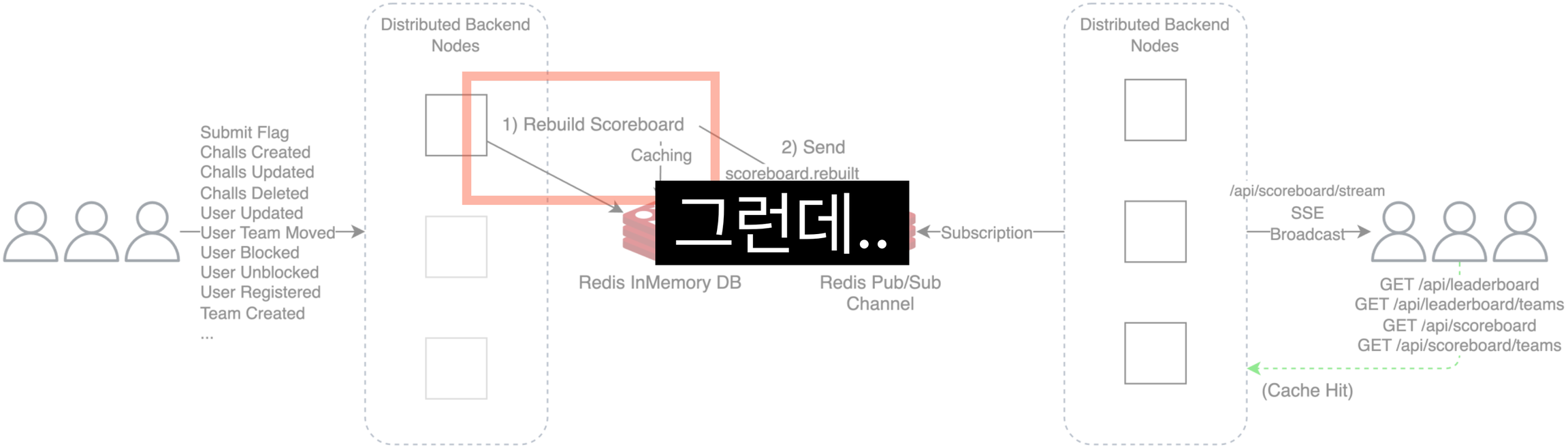
이러한 문제를 해결하기 위해 처음 고안한 구조는 다음과 같음.



한번 캐싱한 뒤 이벤트(scoreboard.rebuilt)를 Publish하기 때문에, 이후 알림 이벤트(scoreboard.rebuilt)를 받은 클라이언트에서의 요청으로 인한 Thundering Herd 문제는 해결할 수 있음.

Thundering Herd 문제 해결하기

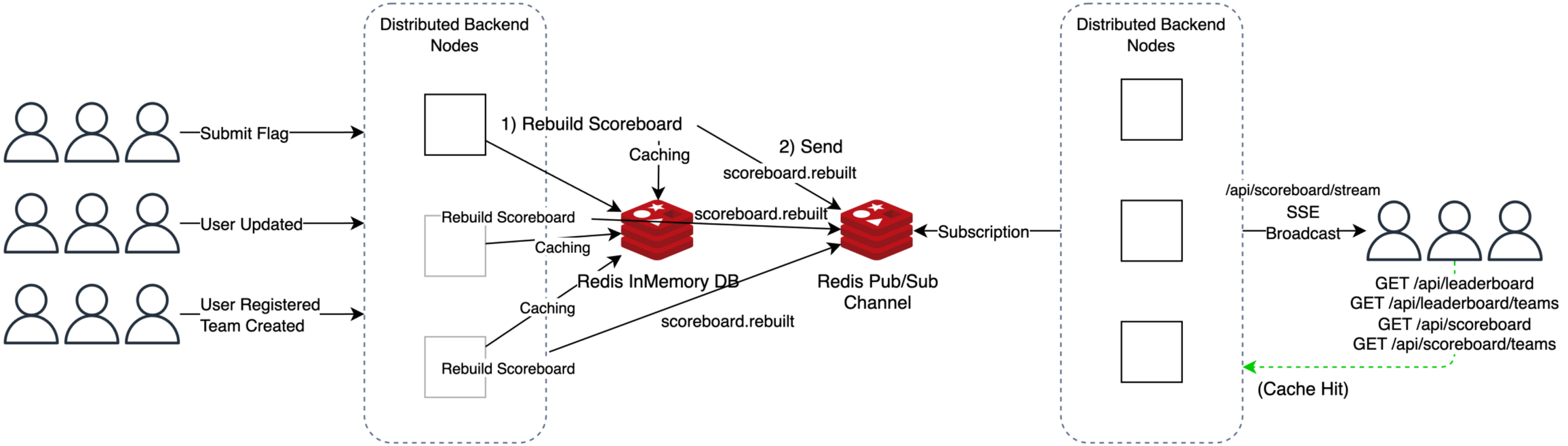
이러한 문제를 해결하기 위해 처음 고안한 구조는 다음과 같음.



한번 캐싱한 뒤 이벤트(scoreboard.rebuilt)를 Publish하기 때문에, 이후 알림 이벤트(scoreboard.rebuilt)를 받은 클라이언트에서의 요청으로 인한 Thundering Herd 문제는 해결할 수 있음.

아직 해결되지 않은 Burst 요청 처리하기

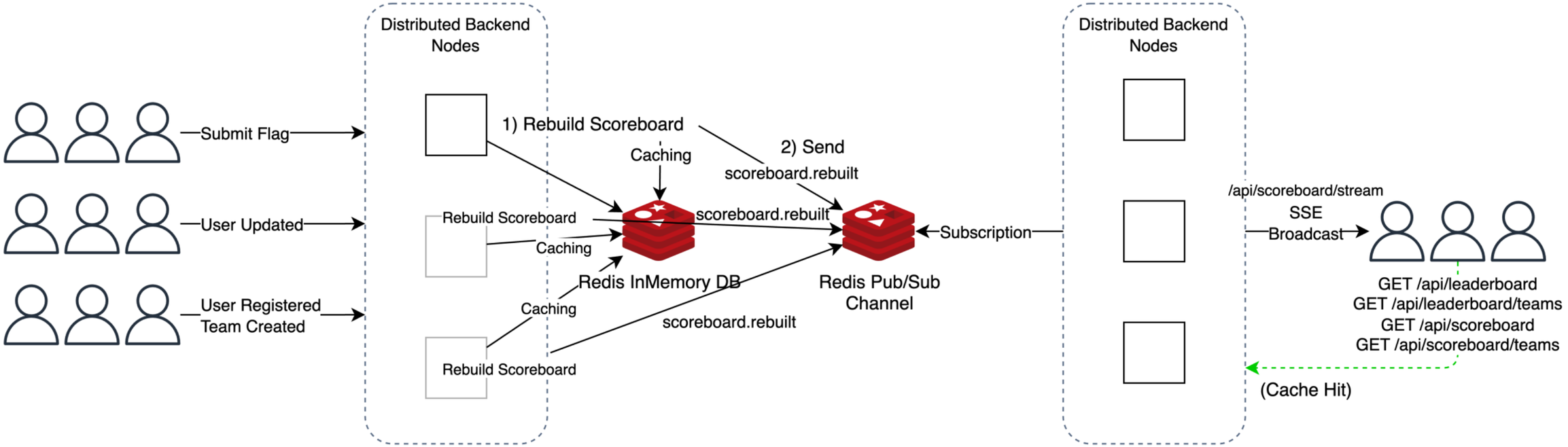
동시간대의 여러 요청이 발생할때, 스코어보드 업데이트에 의한 리빌드 및 캐싱이 필연적으로 발생할 수 밖에 없음.



물론 일반적인 CTF 대회 규모에선 전혀 문제가 되지 않지만, 추후 대규모 플랫폼으로 확장하거나 프로젝트의 규모를 무제한으로 수용할 수 있도록 메인테인하고 있기 때문에 짚고 넘어가야할 부분임.

아직 해결되지 않은 Burst 요청 처리하기

스코어보드 리빌드 시간이 늘어지면서 최악의 경우 결과가 꼬일 수도 있기* 때문에 꽤나 중요한 사안임.



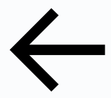
※ 스코어보드 계산에 대한 단일 트랜잭션이나 원자적 처리를 하지 않았음.

아직 해결되지 않은 Burst 요청 처리하기

또 이 문제와는 별개로 스코어보드 캐싱 무효화 및 Pub/Sub 이벤트 전송 로직이 API 핸들러에 커플링되어있어 역할이 명확하지지 않는다는 단점이 있고, 이로 인한 오버헤드와 레이턴시가 있다는 단점도 존재함.

(단일 책임 원칙(=SRP)는 OOP SOLID 원칙에서만 존재하는게 아님, 특히 클린 코드/아키텍처를 지향하는 필자로썬 결점임)

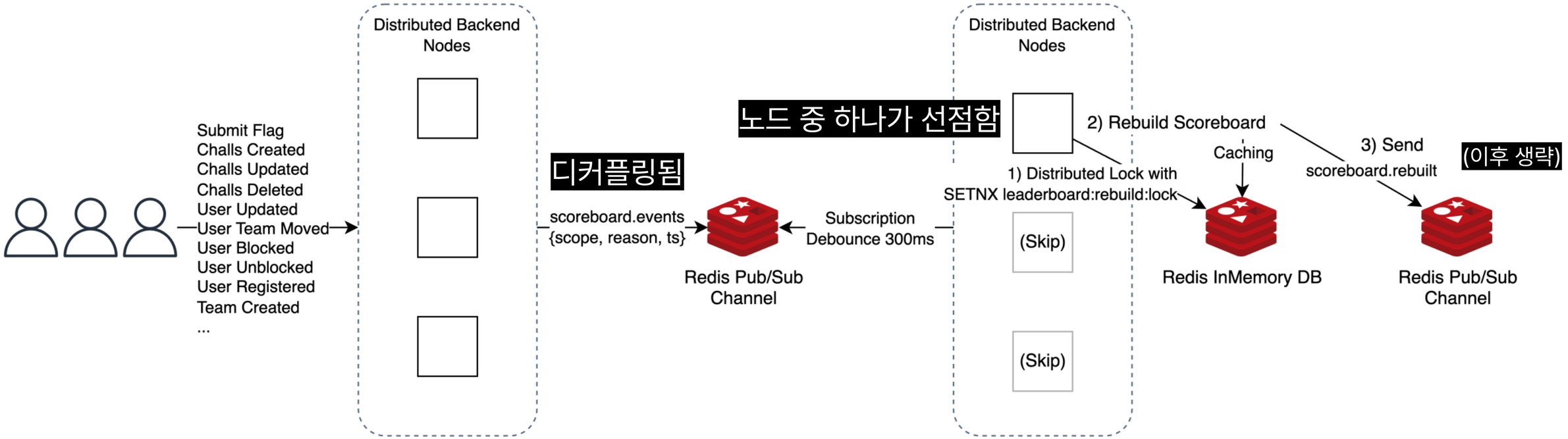
```go
func (h *Handler) SubmitFlag(ctx *gin.Context) {
    // ... (중략)

    if correct {
        h.invalidateTimelineCache()
        h.invalidateLeaderboardCache()
        h.rebuildScoreboard()
        // ... (중략)
    }

    // ... (중략)
}
```



SubmitFlag Handler의 역할과 관련이 크게 없음에도 커플링되어 직접적인 영향을 미침.

아직 해결되지 않은 Burst 요청 처리하기 — 디커플링하기

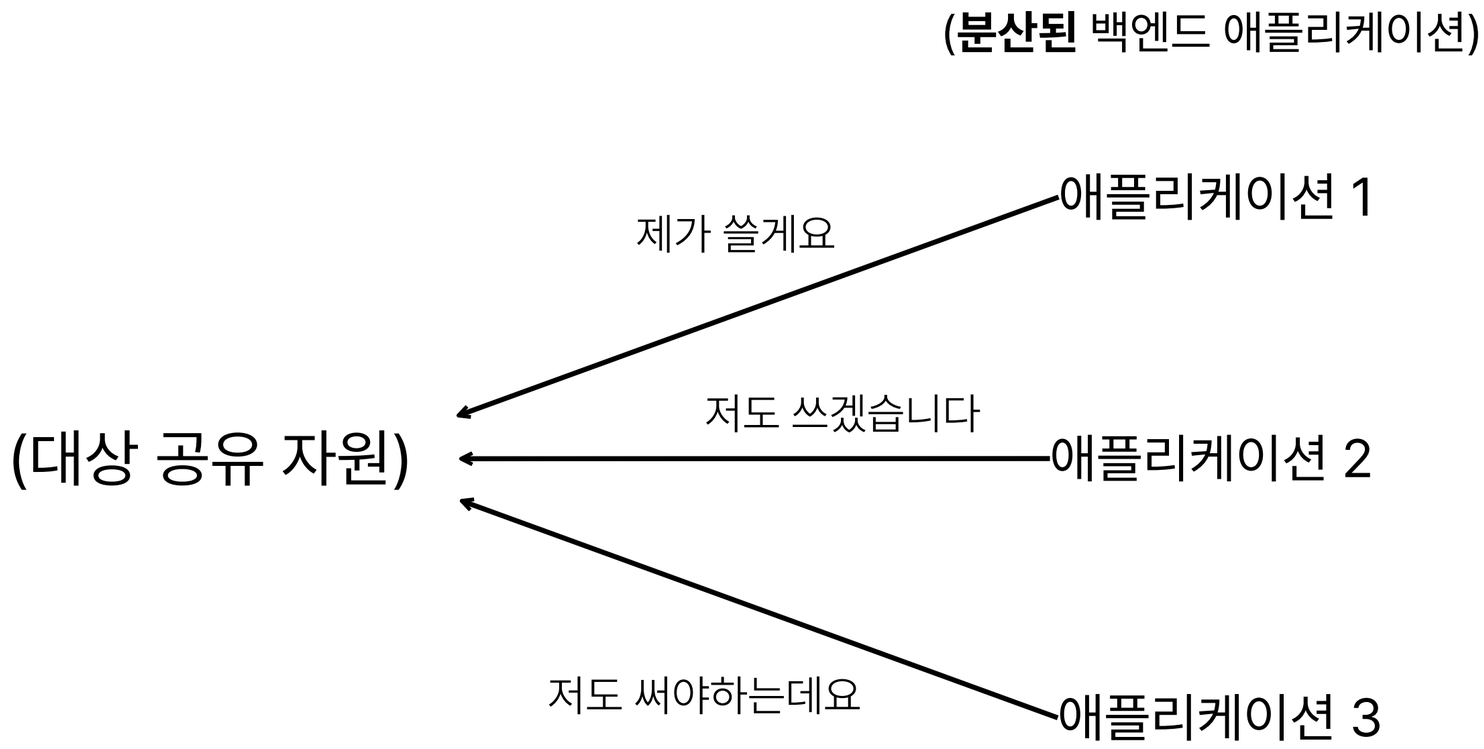
이러한 문제를 해결할 수 있는 가장 적절한 방법이 **디커플링(Decoupling)** 및 디바운스(Debounce) 전략임.



Redis Pub/Sub 채널(scoreboard.events)을 앞단에 하나 더 추가하고, 클라이언트가 스코어보드에 변동을 주는 API 호출 시 스코어보드 계산을 해당 API 핸들러에서 처리하는 대신 scoreboard.events 채널로 전송함. 이후 해당 채널을 Subscribe 하는 **모든 노드들 중 하나만** 선정하여 이벤트 처리 후 기존 scoreboard.rebuilt 채널에 이벤트를 보내는 것. **(디커플링)**

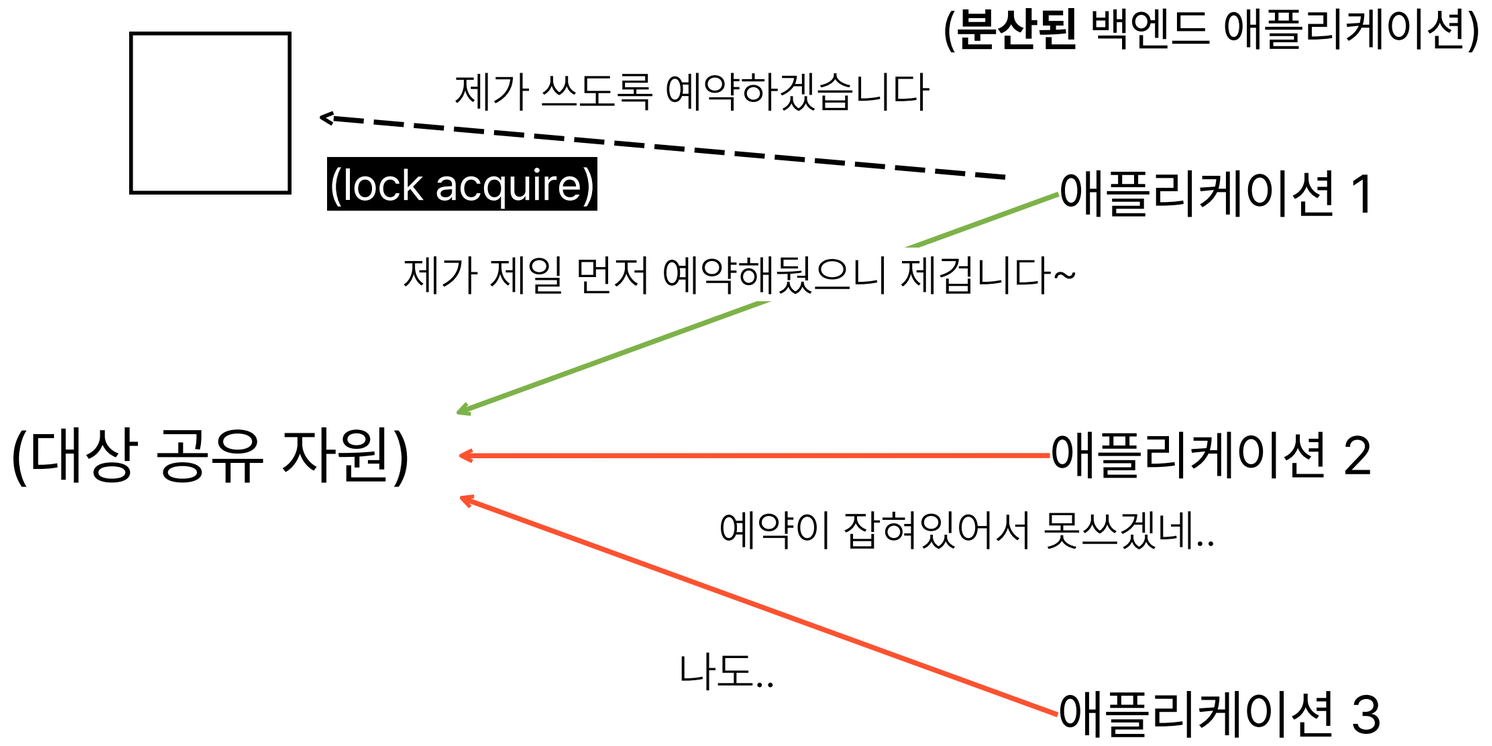
아직 해결되지 않은 Burst 요청 처리하기 — 디커플링을 위한 분산 락

이때 여러 노드 중 하나가 scoreboard.events를 선점하는 방식에선 여러 방법이 있으나, 여기선 **분산 락(Distributed Lock)** 기법을 사용하였음.



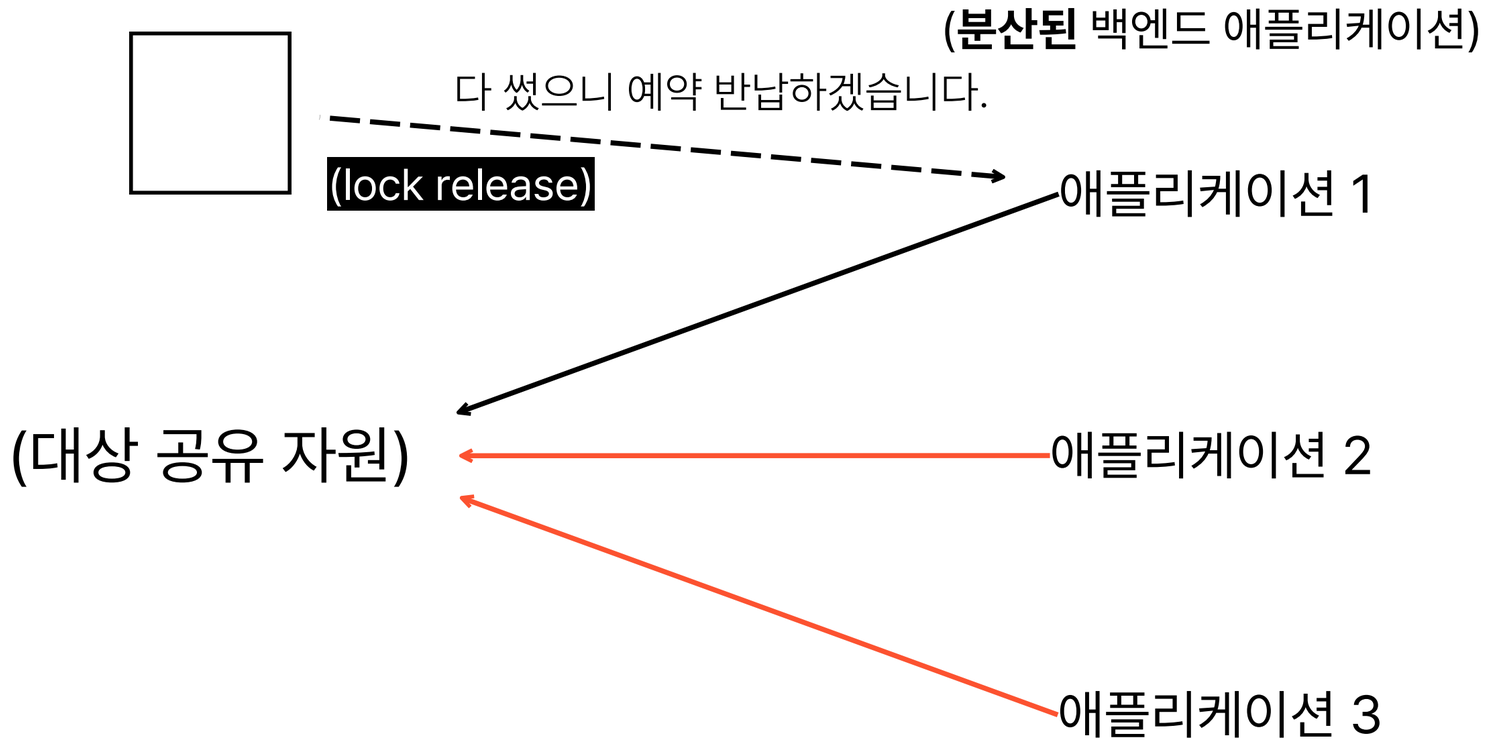
아직 해결되지 않은 Burst 요청 처리하기 — 디커플링을 위한 분산 락

이때 여러 노드 중 하나가 scoreboard.events를 선점하는 방식에선 여러 방법이 있으나, 여기선 **분산 락(Distributed Lock)** 기법을 사용하였음.



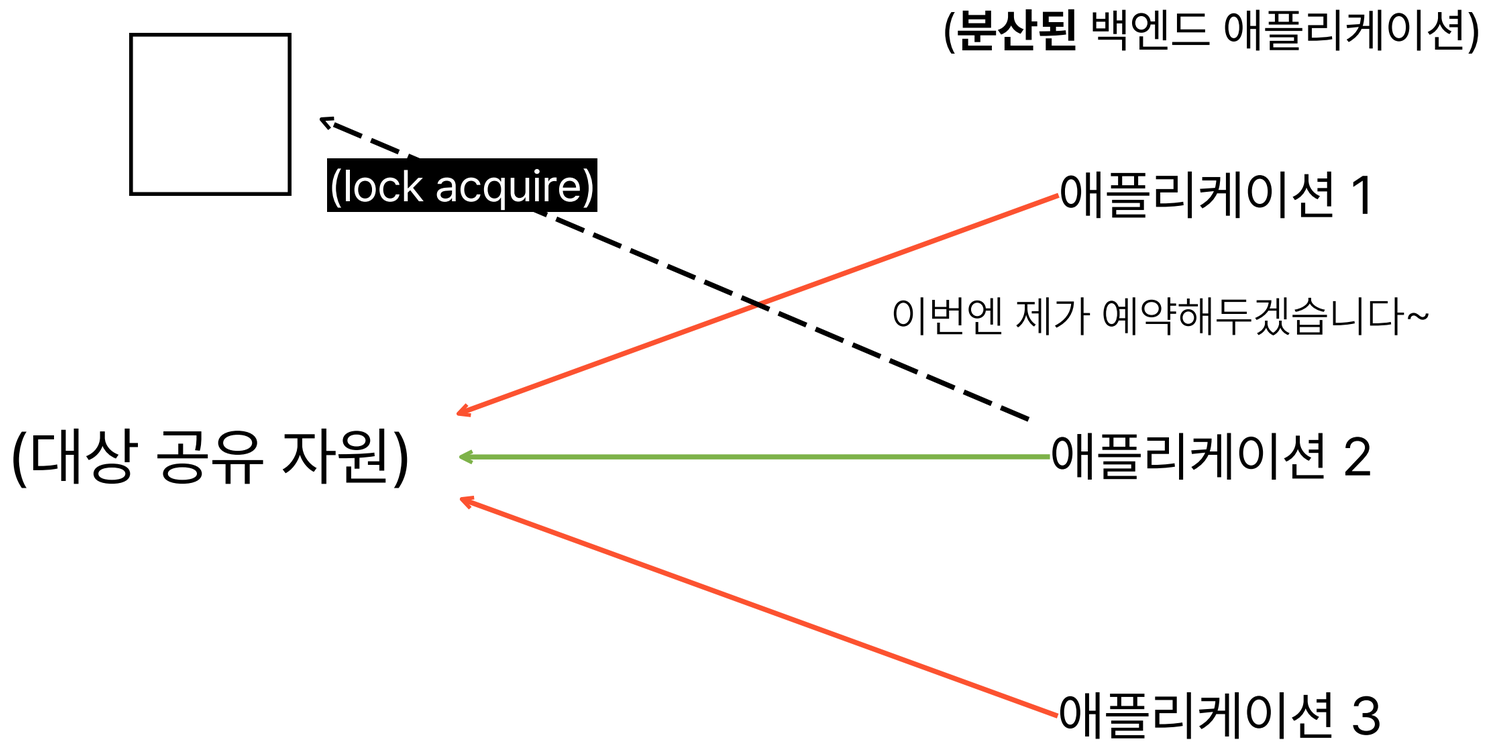
아직 해결되지 않은 Burst 요청 처리하기 — 디커플링을 위한 분산 락

이때 여러 노드 중 하나가 scoreboard.events를 선점하는 방식에선 여러 방법이 있으나, 여기선 **분산 락(Distributed Lock)** 기법을 사용하였음.



아직 해결되지 않은 Burst 요청 처리하기 — 디커플링을 위한 분산 락

이때 여러 노드 중 하나가 scoreboard.events를 선점하는 방식에선 여러 방법이 있으나, 여기선 **분산 락(Distributed Lock)** 기법을 사용하였음.

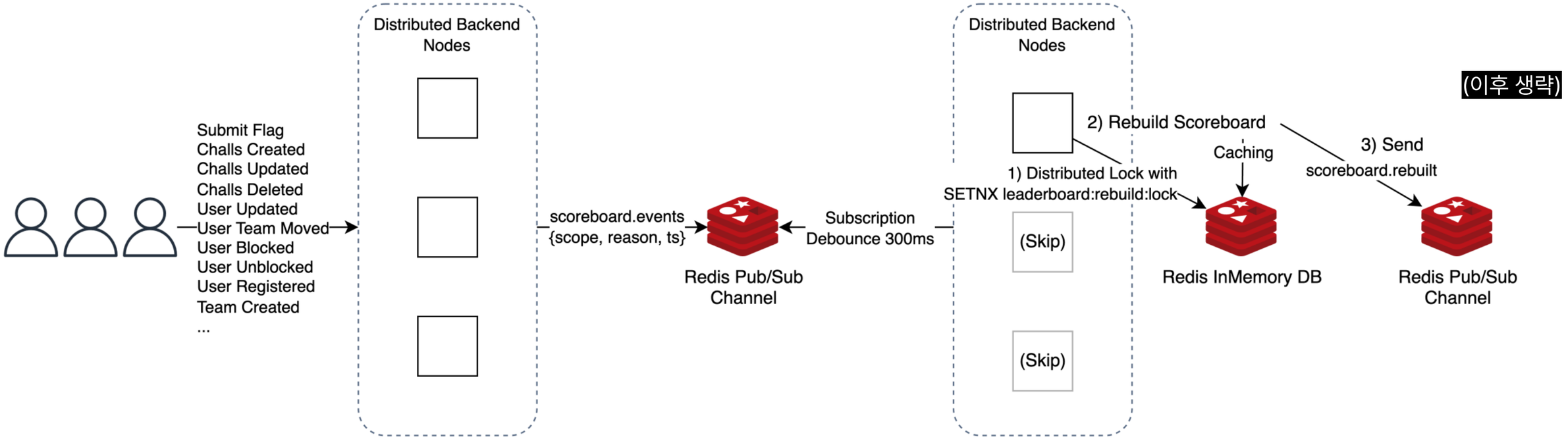


아직 해결되지 않은 Burst 요청 처리하기 — 디커플링을 위한 분산 락

이때 여러 노드 중 하나가 scoreboard.events를 선점하는 방식에선 여러 방법이 있으나, 여기선 **분산 락(Distributed Lock)** 기법을 사용하였음.

아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

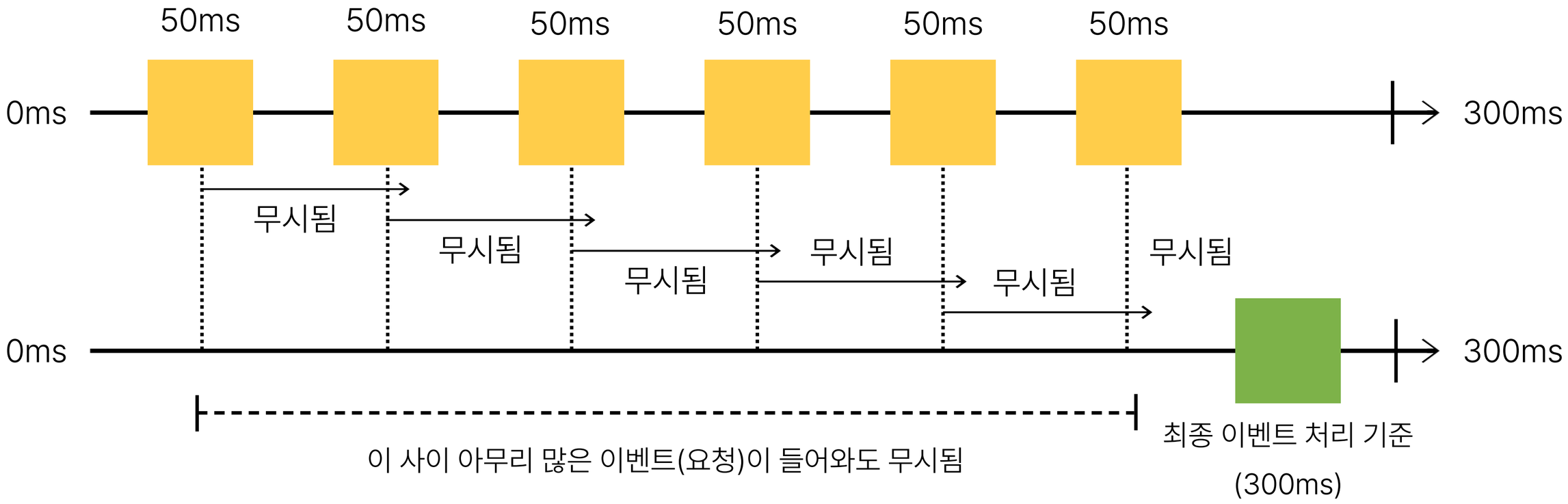
선정된 노드는 이전 커플링 상황에서의 스코어보드 계산, 캐싱 및 최종 이벤트(scoreboard.rebuilt)를 Publish 하는데, 이때 만약 동시간대에 연속적인/Burst한 요청이 들어올 경우, 마찬가지로 많은 리소스 소모가 발생할 수 있음.



이러한 상황에 대비하기 위해 **디바운스(Debounce)** 기법/전략을 사용할 수 있음.

아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

디바운스(Debounce) 기법은 특정 시간 만큼 대기한 뒤 마지막 이벤트(요청) 기준으로 처리하는 최적화 기법임.
(이때 대기할때 들어온, 즉 마지막 이벤트를 제외한 앞선 이벤트는 무시됨)



이는 앞선 분산 락을 통한 디커플링 없이도 구현이 가능하긴 하지만, 디커플링의 장점과 디바운스를 쉽게 구현할 수 있다는 장점으로 같이 사용하도록 구현했음. (요청에 대한, 즉 앞단의 scoreboard.events만 보면 되니 구현이 쉬워짐)

아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

Q 디바운스 기간동안 마지막 이벤트를 제외한 이벤트들은 버려진다면 문제가 생기는 것 아닌가요?

A 일단 기존에 설계 자체가 Pub/Sub 메시지(이벤트)에 어떠한 페이로드도 포함하지 않도록* 설계되었음.

(SSE를 통해 직접 페이로드를 전송하지 않는 이유는 효율성 및 향후 확장성, 그리고 유지보수 차원에서 코드 변경을 최소화하기 위함)

26p 中

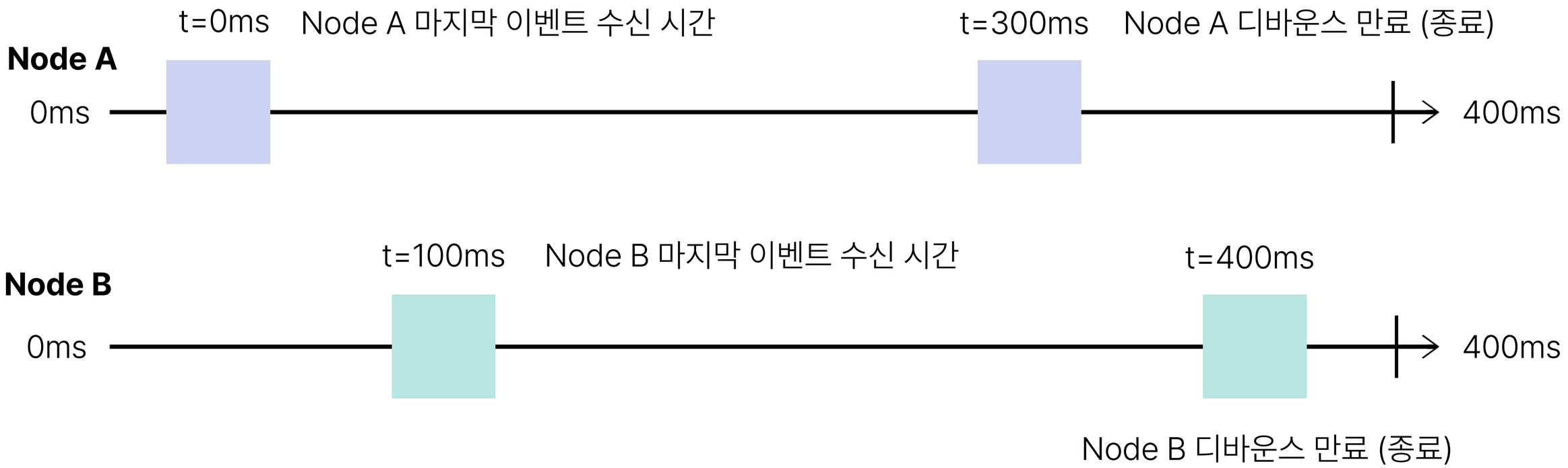
그리고 지금 형태의 구조에선 디커플링의 목적으로 스코어보드 계산 및 캐싱은 디바운스 이후 진행되고, 즉 이러한 디바운스 기간에 버려지는 것들은 비즈니스 로직에 영향을 주지 않음. (여기서 모든 Pub/Sub 이벤트는 단순히 알림 수준임)

※ 디버깅을 위한 Timestamp 등의 메타데이터 제외.

아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

Refs 분산 락과 디바운스를 도입하여 Burst 요청을 처리한다고 해도, **"해결"**이 아닌 **"완화"** 한다는 표현을 사용해야함.

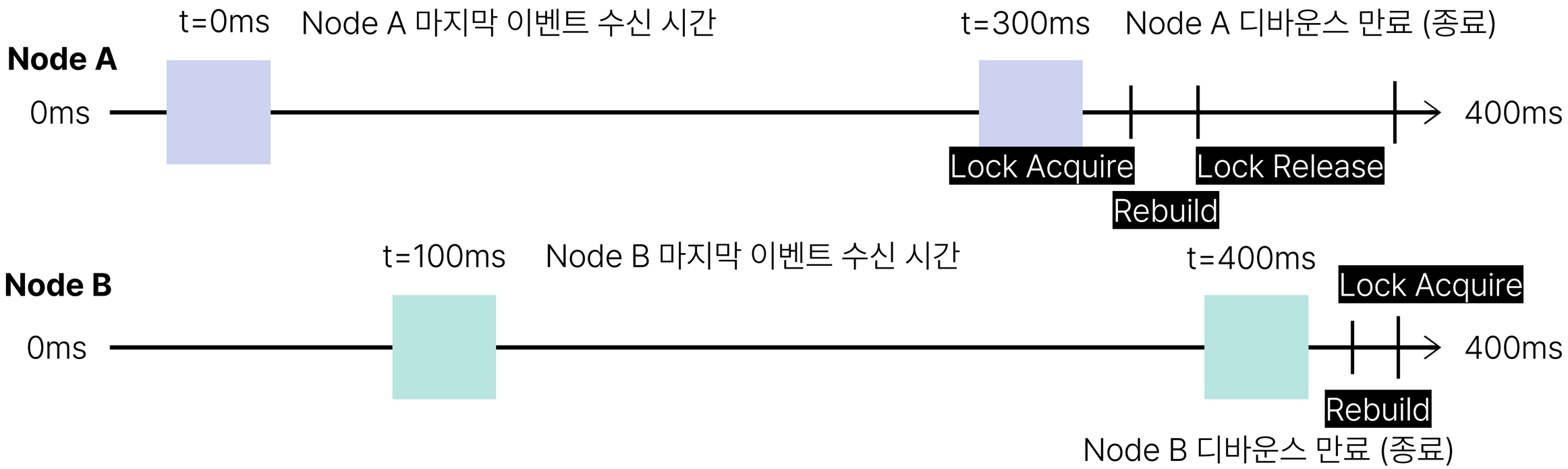
하나의 예시로 다음과 같은 시나리오를 생각해볼 수 있음. 두 백엔드 노드(A, B)가 300ms 주기로 디바운스 하는 것은 동일하지만, 어떠한 변수(네트워크 레이턴시 등등)로 인해 B 노드가 100ms 지연되었다고 가정할 수 있음.



아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

Refs 분산 락과 디바운스를 도입하여 Burst 요청을 처리한다고 해도, **"해결"**이 아닌 **"완화"** 한다는 표현을 사용해야함.

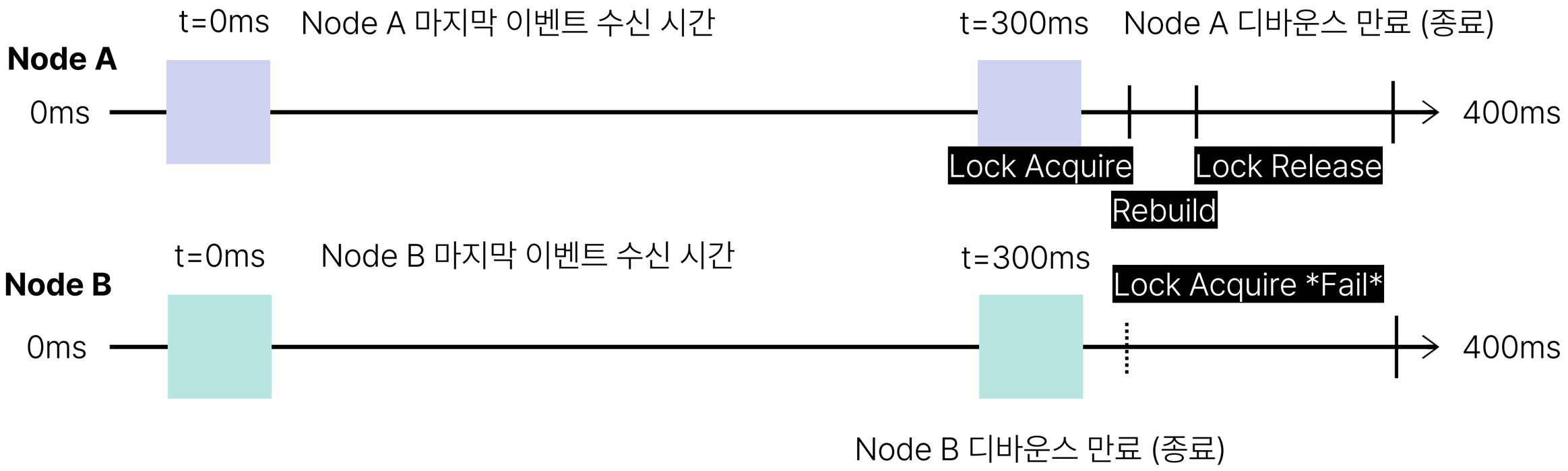
이때 노드 A가 락을 잡은지 10ms만에 처리가 끝났다면, 노드 B가 400ms 시점에서 락을 잡아버릴 수 있음. 이 경우 300ms 주기로 스코어보드 리빌드를 한다는 이론과 맞지 않게 300ms 이후 100ms 안에 리빌드가 또 발생할 수 있음.



아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

Refs 분산 락과 디바운스를 도입하여 Burst 요청을 처리한다고 해도, **"해결"**이 아닌 **"완화"** 한다는 표현을 사용해야함.

만약 노드 B의 100ms 지연이 없었다면 아래와 같이 정상적으로 300ms 주기로 동작했어야 할 것임. 그래서 이러한 변수로 인해 디바운스가 Burst 요청 처리를 완전히 "해결" 할 수 없고 "완화" 할 수 있다는 표현이 맞는 이유임.



아직 해결되지 않은 Burst 요청 처리하기 — 디바운스(Debounce) 전략

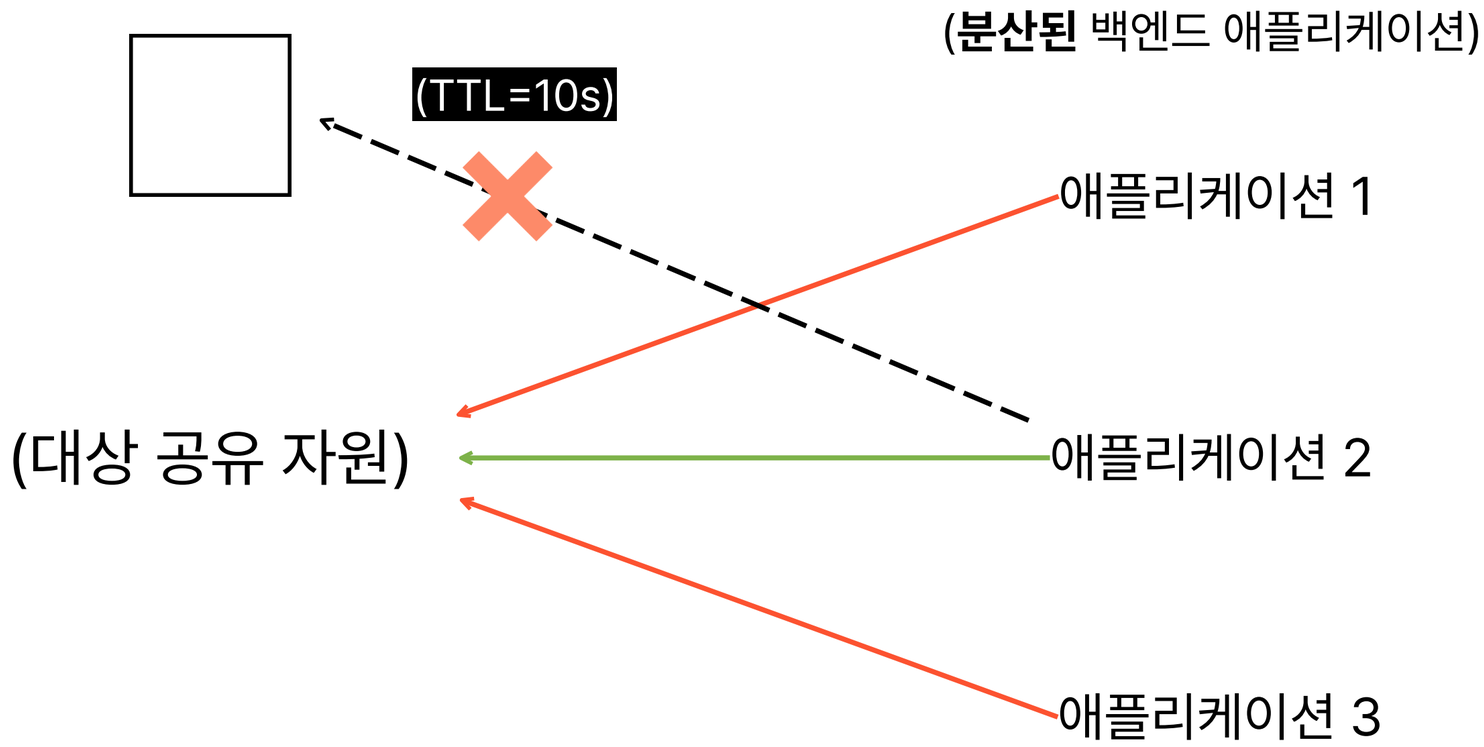
Refs 분산 락과 디바운스를 도입하여 Burst 요청을 처리한다고 해도, **"해결"**이 아닌 **"완화"** 한다는 표현을 사용해야함.

물론 이것 조차 완화하려면 처리 직후 쿨다운 등을 도입하거나, 마지막 계산 시각을 last_rebuilt 키 등으로 저장하고 처리하는 방법 등이 있을 수 있지만, 거기까진 너무 복잡하다고 판단하였고 이 정도로 만족하기로 타협을 봤음..

(이에 대한 피드백은 언제나 환경)

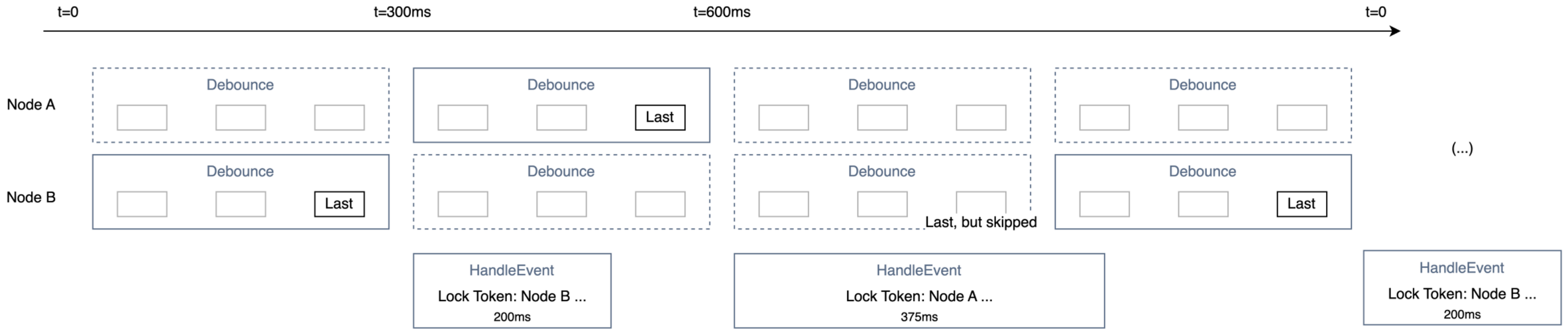
아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

락을 잡고난 뒤 어떠한 변수로 인해 락이 영구적으로 반환(Release)되지 않을 수 있음. 이러한 상황을 막기 위해 TTL을 적용함.



아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

분산 락의 TTL은 적절한 타협점이 필요한데, 너무 짧다면 중복으로 실행되거나 리소스 소모가 많아진다는 단점이 존재하고 너무 길다면 장애 발생 시 처리 시간이 길어진다는 트레이드오프가 존재함.



위 다이어그램은 분산 락의 동작 과정을 나타냄. 락이 너무 길어질 경우 처리 시간이 길어짐을 나타냄. (토큰은 후술)

또한 위와 같이 문제 없이 진행되는 경우는 네트워크 레이턴시 등의 변수가 없다는 가정하에

락 TTL ≥ 최대 처리 시간이 성립할때 유효함

아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

통상적으로 이러한 상황에서 락 TTL은 아래의 공식을 따르면 됨.

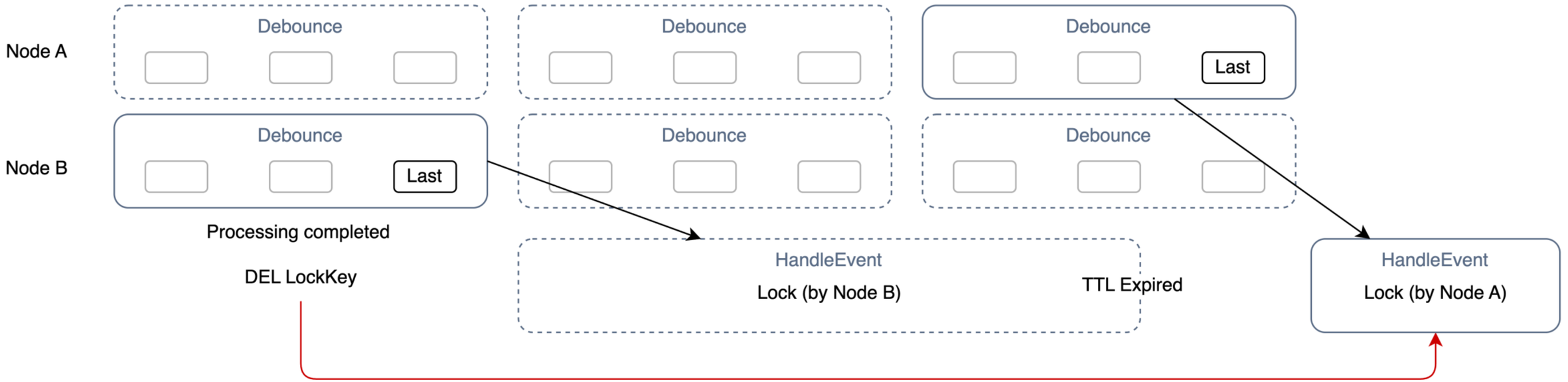
- TTL ≥ (handleEvent* 최대 실행 시간 p95* + 네트워크 지연 + clock skew*)
- (간단하게) handleEvent 최대 실행 시간 p95의 2~3배 정도

테스트/스테이징 환경에서 handleEvent의 p95가 2초 내외였으나, 규모가 다른 프로덕션 환경에서 테스트는 확인되지 않았기 때문에 이 프로젝트에선 10초를 기본 값으로 정하고 있음.

※ handleEvent : 락을 잡고 스코어보드 처리, 캐싱 및 락 반환과 rebuilt Pub/Sub 채널 메시지 발행 함수
※ p95 : 데이터를 크기순으로 정렬했을 때 전체의 95%가 해당 값보다 작거나 같고, 나머지 5%는 그보다 큰 지점을 의미하는 지표.
※ Clock Skew : 분산 환경이나 네트워크에 연결된 여러 장치의 Clock이 서로 일치하지 않고 시간 차이가 나는 현상. 대부분 하드웨어적 요인

아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

handleEvent 처리가 끝나기 전, 즉 락이 반환되기 이전의 시점에서 락 TTL이 끝날 경우 아래와 같은 문제가 발생할 수 있음.

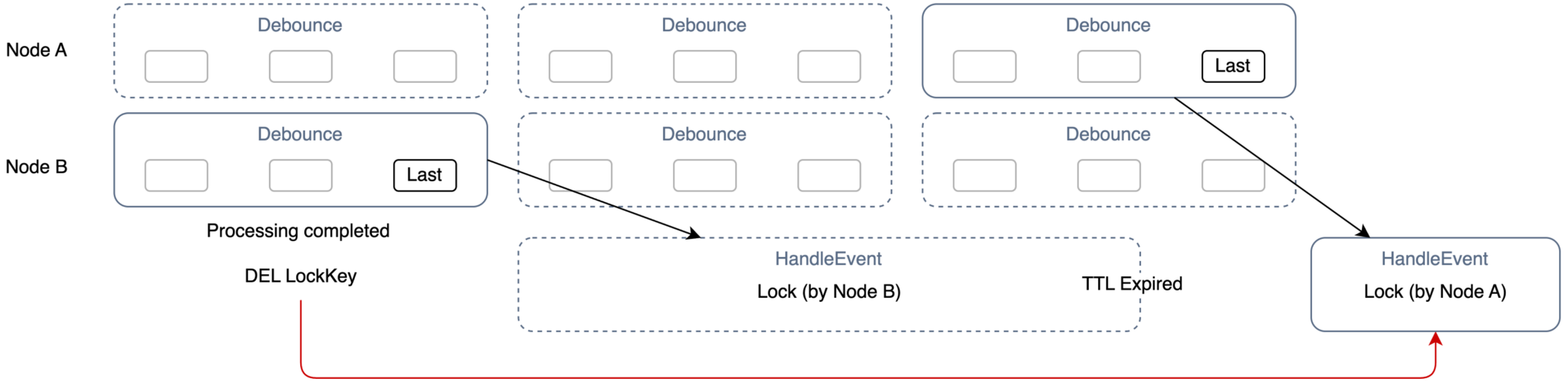


노드 B가 락을 가져갔는데, 처리가 오래걸리 TTL을 초과됨. 이후 새로운 디바운스 기간이 지나 노드 A가 락을 잡았다고 가정.

이때 노드 B의 HandleEvent가 그제서야 처리를 끝마쳐 락을 삭제하게 됨.

아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

이 경우 노드 B는 현재 존재하는 락이 노드 A의 락인지 알 수 없으며, 무지성으로 해당 락을 삭제시킬 수 있음.



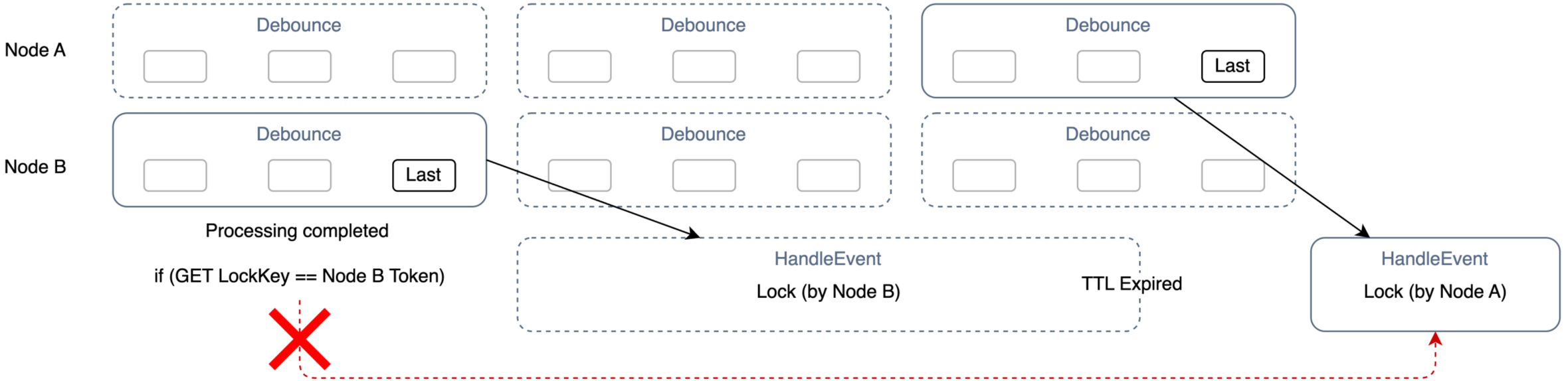
이 경우 앞선 "짧은 시간 내에 여러번의/중복된 계산"이 실행될 수 있음.

(물론 처리 시간이 락 TTL을 넘기는 경우는 거의 없겠지만, 최악의 경우를 언제나 대비해야함)

※ 락 TTL이 만료되었을때 실행중인 HandleEvent를 종료시키면 안되냐는 질문이 있을 수 있지만, 서로 다른 고루틴(병렬)로 동작하기 때문에 어려움.

아직 해결되지 않은 Burst 요청 처리하기 — TTL과 분산 락의 토큰

이를 해결하기 위해 락 키에 노드에 대한 임의의 토큰 전략을 사용하고, 락킹 시 생성되는 토큰과 동일해야 릴리즈 될 수 있도록 함.



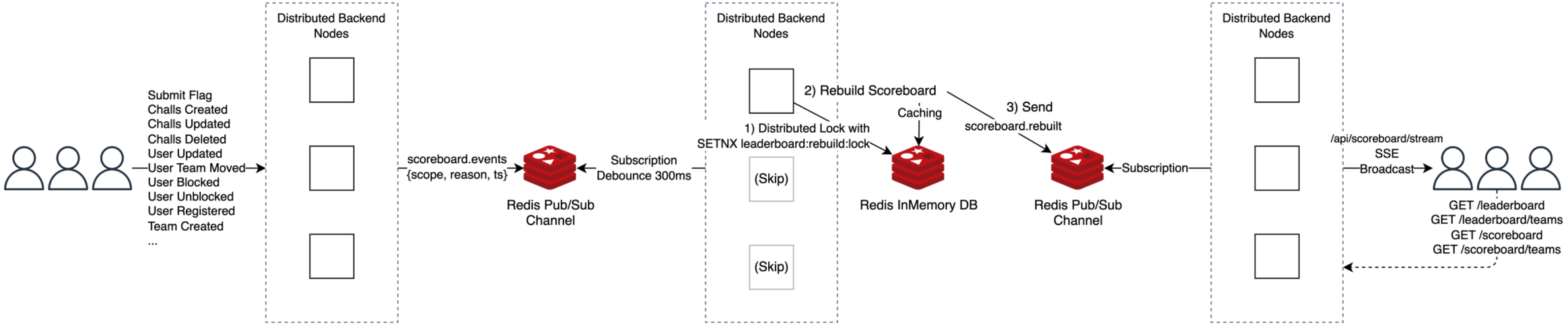
이러한 노드 B 입장에선 현재 존재하는 락 키가 본인의 락 키가 아니기 때문에 무시하도록 할 수 있음.

(이에 대한 구현 방식 및 Redis 원자성 고려는 구현 섹션에서 서술)

※ 이때 토큰은 락 획득(acquire) 시 랜덤으로 지정되며, Failover로 현재 Unix Time을 기준으로 생성하도록 함.

아직 해결되지 않은 Burst 요청 처리하기

최종적으로 고안된 아키텍처는 아래와 같음. (자세한 이미지 내용은 블로그 참조)



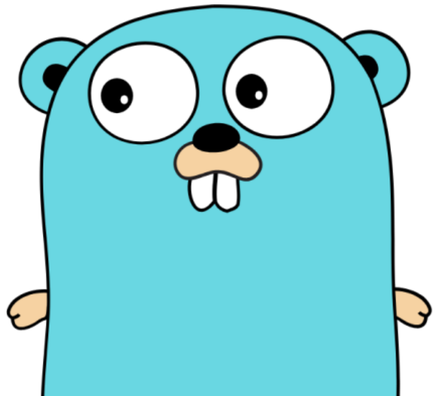
When the cache is empty, many clients can hit the leaderboard API at the same time and trigger the same expensive recomputation multiple times.
With SSE, these bursts are more synchronized, so this "thundering herd" effect becomes more likely.

We mitigate this by debouncing events, letting only one node rebuild the cache under a Redis lock, and broadcasting an update only after the cache is refreshed.

구현하기 in Go

프로젝트에서 사용되는 언어는 **Go 언어**임. 매우 뛰어난 퍼포먼스와 동시성 처리의 강력함, 직관적인 문법 등으로 백엔드나 특히 인프라/클라우드 환경에서 자주 사용되는 언어임. (Kubernetes와 CNCF 생태계의 공식적인 개발 언어)

Go 언어에서의 백엔드 개발 프레임워크인 Gin과 PostgreSQL ORM을 위한 Bun을 사용함. 원래는 NestJS가 주력이였으나, 너무 무겁고 오버 엔지니어링이라 판단하여 Lightweight를 추구했던 프로젝트에서 적합하지 않다고 판단, 배제하였음.



본 섹션에선 분량 상 핵심적인 비즈니스 로직만 설명하고 불필요한 코드는 생략함.

관련 소스코드 및 PR은 다음 링크에서 참조.

- https://github.com/nullforu/smctf
- https://github.com/nullforu/smctf/pull/43

구현하기 in Go

먼저 스코어보드에 영향을 주는 경우는 올바른 플래그 제출, 유저 차단/해제, 유저 정보 변경, 문제 CUD, 회원가입, 팀 생성 등이 스코어보드 변동에 영향을 줌. 여러 핸들러에서 같은 기능을 처리해야 하기 때문에 공통된 헬퍼 함수를 추가하였음.

```go
// internal/http/handlers/handler.go

func (h *Handler) notifyScoreboardChanged(ctx context.Context, reason string) {
    h.invalidateLeaderboardCache()
    h.invalidateTimelineCache()
    h.publishScoreboardEvent(ctx, reason)
}
```

캐시 무효화

scoreboard.events 이벤트 발행 (디바운스 기준 이벤트)

```go
func (h *Handler) publishScoreboardEvent(ctx context.Context, reason string) {
    event := realtime.ScoreboardEvent{
        Scope:  "all",
        Reason: reason,
        TS:     time.Now().UTC(),
    }

    payload, err := json.Marshal(event)
    if err != nil {
        return
    }

    _ = h.redis.Publish(ctx, "scoreboard.events", payload).Err()
}
```

구현하기 in Go

```go
// internal/realtime/scoreboard_bus.go

type ScoreboardBus struct {
    redis    *redis.Client
    cfg      config.Config
    score    ScoreboardReader
    logger   *logging.Logger
    hub      *SSEHub
    debounce time.Duration
    lockTTL  time.Duration
    trigger  chan string
}

func (b *ScoreboardBus) Start(ctx context.Context) {
    pubsub := b.redis.Subscribe(ctx, scoreboardEventsChannel)
    rebuilt := b.redis.Subscribe(ctx, scoreboardRebuiltChannel)

    go b.run(ctx, pubsub, rebuilt)
}

func (b *ScoreboardBus) run(ctx context.Context, pubsub *redis.PubSub, rebuilt *redis.PubSub) {
    defer func() {
        if err := pubsub.Close(); err != nil {
            b.logger.Warn("leaderboard pubsub close", slog.Any("error", err))
        }

        if err := rebuilt.Close(); err != nil {
            b.logger.Warn("leaderboard rebuilt close", slog.Any("error", err))
        }
    }()

    // ... (중략)
}
```

이제 첫 이벤트(scoreboard.events)가 전송되는 채널을 Subscribe 하고

디바운스 및 분산 락을 적용하는 로직을 작성해야함.

이는 별도의 마이크로서비스로 분산할 수 있지만, 유지보수 포인트가 늘어날 것 같아 패스.

이때 Goroutine을 사용함.

※ Goroutine은 Go에서 제공하는 매우 가벼운 병렬 처리 기법인데, 초경량 스레드라고 생각하면 됨.

구현하기 in Go — scoreboard.events Subscribe

스코어보드에 변동이 생길 경우 무효화와 함께 이벤트가 발행되는 채널로, 메시지가 있을 경우 가져와 적절하게 처리하면 됨.

이때 해당 채널을 Subscribe 하는 Goroutine 내에서 모든 처리를 하면 병목이 발생할 수 있고, 그렇다고 메시지마다 Goroutine을 만드는 방식은 메모리/GC에 상당한 부하를 발생시킬 수 있으므로 별도의 Trigger 채널을 만들어 분리하는 방식을 사용함. (이전의 ScoreboardBus 구조체 내의 trigger가 그 정의임.)

```
go func() {
    ch := pubsub.Channel()
    for {
        select {
        case <-ctx.Done():
            return
        case msg, ok := <-ch:
            if !ok {
                return
            }

            select {
            case b.trigger <- msg.Payload:
            default:
            }
        }
    }
}()
```

이후 디바운스 로직은 Time과 이 Trigger 채널을 기반으로 하는 State Machine 형태라 비동기 처리를 할 필요가 없음.

※ 서로 다른 Goroutine에서 서로 데이터를 송수신하기 위해 사용되는 기능이 채널(Channel)인데, 여기선 Trigger 용도로 사용

구현하기 in Go — scoreboard.events Subscribe

디바운스 구현을 위해 time.Timer를 사용하며, timer.C 채널은 타이머가 만료되었음을 알려주는 수신 전용 채널이니 활용 가능.

```
var (
    timer       *time.Timer
    lastPayload string
)

for {
    select {
    case <-ctx.Done():
        if timer != nil {
            timer.Stop()
        }
        return
    case payload := <-b.trigger:
        lastPayload = payload
        if timer == nil {
            timer = time.NewTimer(b.debounce)
            continue
        }

        if !timer.Stop() {
            select {
            case <-timer.C:
            default:
            }
        }

        timer.Reset(b.debounce)
```

```
    case <-func() <-chan time.Time {
        if timer == nil {
            return nil
        }
        return timer.C
    }():
        if timer != nil {
            timer.Stop()
            timer = nil
        }

        b.handleEvent(ctx, lastPayload)
    }
}
```

좀 더 세부적으로 살펴보자면..

구현하기 in Go — scoreboard.events Subscribe

```
case payload := <-b.trigger:
    lastPayload = payload
    if timer == nil {
        timer = time.NewTimer(b.debounce)
        continue
    }

    if !timer.Stop() {
        select {
        case <-timer.C:
        default:
        }
    }

    timer.Reset(b.debounce)
```

Trigger 채널에 이벤트가 수신될 경우 실행되는데,

마지막 이벤트 페이로드만 저장하고 타이머를 디바운스 만큼 초기화함.

이 경우 이벤트가 절대 멈추지 않고 계속 수신된다면 영원히 핸들링되지 않을 수 있겠지만,

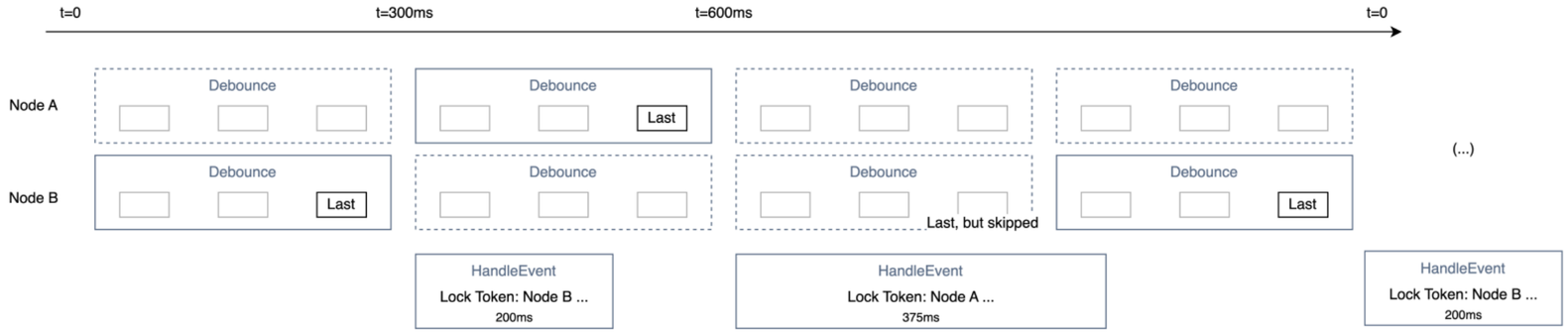
플랫폼 특성상 + 300ms의 짧은 디바운스 간격에선 그럴 경우는 없으니 문제가 되지 않음. (낙관적)

※ 원래 디바운스가 "300ms 마다 핸들링" 한다라는 전제였지만, 이는 이론상이고 실제 구현에선 복잡성 및 타협점을 고려하여 이렇게 구현하였음.

구현하기 in Go — scoreboard.events Subscribe

이후 만약 300ms동안 이벤트가 들어오지 않는다면 타이머가 만료되고, 락을 잡은 하나의 노드가 스코어보드를 처리할 수 있음.

```
case <-func() <-chan time.Time {
    if timer == nil {
        return nil
    }
    return timer.C
}():
    if timer != nil {
        timer.Stop()
        timer = nil
    }

    b.handleEvent(ctx, lastPayload)
}
```



여기까지의 로직이 Burst 요청 처리를 위한 디바운스 구현 부분임

분산 락을 잡고 릴리즈 까지의 로직은 handleEvent에 포함됨.

※ 중간에 익명 함수는 타이머가 nil일 경우 timer.C를 사용하면 Panic이 발생하므로 이에 대한 처리임. nil case는 실행되지 않음.

구현하기 in Go — handleEvent

이제부터 본격적으로 스코어보드를 처리하는 함수로, 디바운스 이후 단 하나의 노드만이 처리 후 캐싱,
그리고 scoreboard.rebuilt 채널에 이벤트를 발행하는 비즈니스 로직을 구현함.

분산 락 구현이 핵심적이며, TTL 만료 이후 다른 노드의 토큰을 삭제하지 않도록 임의의 토큰을 사용하도록 함.

구현하기 in Go — handleEvent

```go
func (b *ScoreboardBus) handleEvent(ctx context.Context, payload string) {
    locked, token := b.acquireLock(ctx)
    if !locked {
        return
    }

    ctx, cancel := context.WithTimeout(ctx, 10*time.Second)
    defer cancel()

    if err := b.rebuildCaches(ctx); err != nil {
        b.logger.Warn("leaderboard rebuild failed", slog.Any("error", err))
        b.releaseLock(ctx, token)
        return
    }

    b.releaseLock(ctx, token)

    _ = b.redis.Publish(ctx, scoreboardRebuiltChannel, payload).Err()
}
```

락 Acquire (습득)

(이미 락이 있다면 함수 실행 안함)

락 Release (반환)

※ Refs 1. rebuildCaches(Context) 함수는 따로 살펴보지 않겠음. 스코어보드 처리 후 캐싱하는 로직이 포함됐다고 생각하면 됨.
※ Refs 2. Context 타임아웃(TTL과 별개)이 락을 Release 하기 전 아웃될 수 있지만 이러한 상황을 위해 TTL을 걸었으니 문제 없음.
※ Refs 3. 더욱 더 안정성을 원한다면 Lock TTL ≥ ctx timeout + 여유 등으로 살짝의 여유를 주는 것도 좋음.

구현하기 in Go — handleEvent

```
func (b *ScoreboardBus) acquireLock(ctx context.Context) (bool, string) {
    token := randomToken()
    ok, err := b.redis.SetNX(ctx, scoreboardLockKey, token, b.lockTTL).Result()
    if err != nil {
        b.logger.Warn("leaderboard lock error", slog.Any("error", err))
        return false, ""
    }

    return ok, token
}

func (b *ScoreboardBus) releaseLock(ctx context.Context, token string) {
    if token == "" {
        return
    }

    const script = `if redis.call("get", KEYS[1]) == ARGV[1] then return redis.call("del", KEYS[1]) else return 0 end`
    _, _ = b.redis.Eval(ctx, script, []string{scoreboardLockKey}, token).Result()
}
```

락을 Acquire(습득, 획득)하고 Release(반환)하는 로직임.

구현하기 in Go — handleEvent

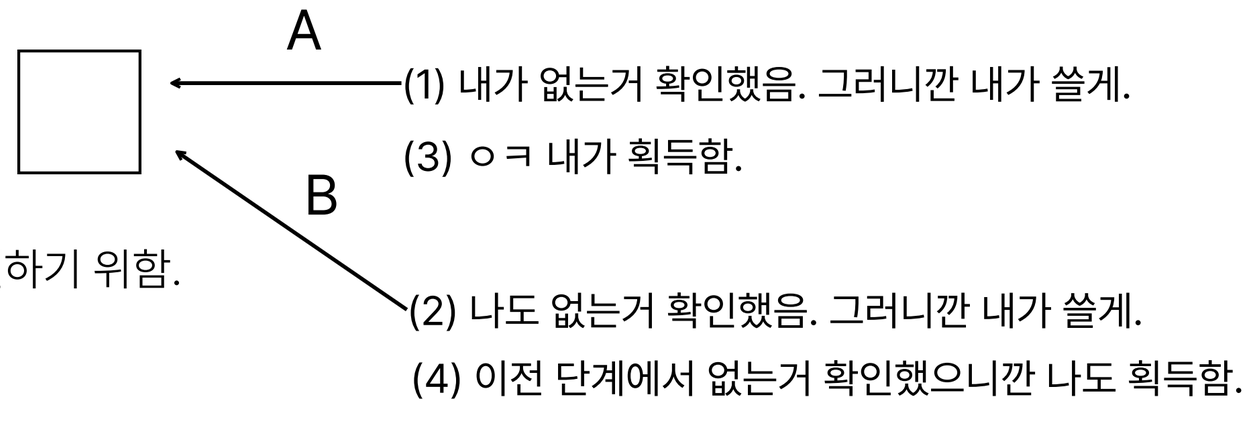
두 함수의 로직을 의사 코드로 표현한다면 다음과 같음.

Acquire

```
SetNX(key, token, TTL)
```

Redis에서 **SetNX**는 **set if not exists**의 약자로, 지정한 키가 존재하지 않을때만 값을 저장하는 명령어임. 락 획득 구현에 부합함.

명령어 자체가 **원자성(Atomicity)**을 지니고 있기에 큰 문제가 없음.



동시간대나 레이턴시 등으로 충분히 발생할 수 있는 문제임.
1 → 3 또는 2 → 4 순서와 같이 중간에서 꼬이는 문제를 해결하기 위함.

(원자성이 A의 작업을 한번에 하도록 보장함)

※ 원자성은 더 이상 쪼갤 수 없는 가장 작은 성질을 의미하는데, DB에선 단일 트랜잭션의 핵심임. 모두 성공하거나 아예 실행되지 않아야 하는 특성.
그리고 이를 처리할땐 다른 명령어가 개입할 수 없음.

구현하기 in Go — handleEvent

두 함수의 로직을 의사 코드로 표현한다면 다음과 같음.

Release

```
if get(key) == token then
    del(key)
end
```

하지만 Release의 경우 자신의 토큰이 맞는지 확인하는 Condition이 추가되고, 참이라면 해당 키를 삭제하는 로직을 가지고 있음.
하지만 이를 원자적으로 처리할 단일 명령어는 Redis에서 제공해주지 않음.

이를 여러 명령어로 조합할 경우 정합성/일관성이 보장되지 않기 때문에 문제가 발생함. (거의 발생하진 않으나 최악의 시나리오)

구현하기 in Go — handleEvent

두 함수의 로직을 의사 코드로 표현한다면 다음과 같음.

Release

```
if get(key) == token then
    del(key)
end
```

1. 노드 A가 락 획득 (Token = A)
2. 해당 락 TTL 만료, 이와 동시에(TTL에 의해 삭제되기 전) Release가 실행될때 GET 결과가 노드 A 락이라 정상 회수(DEL) 예정.
3. TTL에 의해 기존 락이 삭제되고 노드 B가 락 획득 (Token = B)
4. 노드 A가 늦게 DEL 실행, GET 시점에선 노드 A의 토큰이 확인되므로 정상적으로 DEL이 수행됨.

원래라면 2 → 4가 일관되게 이루어져야 하지만, 원자성이 보장되지 않았으므로 그 중간에 3번이 끼어든 것임.

그 결과 노드 B의 락이 삭제되는 문제 발생. → 이를 어떻게 해결할 수 있을까?

※ TTL에 의해 락이 삭제되는 것과 handleEvent에 의해 릴리즈되는 것은 별개로 동작하기 때문에 가능한 시나리오임.

구현하기 in Go — handleEvent

Redis에서 원자성을 위해 단일 트랜잭션을 구현하기 위한 방법엔 크게 2가지가 존재함.

1. MULTI / EXEC

MULTI로 트랜잭션 시작을 알리고, 명령어들을 스택에 쌓은 뒤 EXEC로 한번에 실행하는 방식.

얼핏 보기엔 일반적인 DBMS의 트랜잭션 처리 방식(BEGIN, COMMIT)과 비슷해보이나, 이는 단순히 스택에 쌓고 실행하는 것이기 때문에 롤백 등의 기능이 없음. (전혀 다른 기능)

또한 이번 구현의 핵심인 Condition(조건) 분기를 처리하지 못함. → 배제

구현하기 in Go — handleEvent

Redis에서 원자성을 위해 단일 트랜잭션을 구현하기 위한 방법엔 크게 2가지가 존재함.

2. Lua 스크립팅

Redis 서버 내부에서 동작하는 스크립트 기능으로, 일반적으로 권장되는 원자성 보장 방식.

스크립트가 실행되는 동안 Redis 서버가 블로킹(Blocking) 상태가 되어 다른 클라이언트의 명령이 절대 중간에 실행될 수 없음.

오히려 이 방식이 BEGIN/COMMIT과 흡사하게 All-or-Nothing 특성을 지니며, 분기를 지원하기 때문에 가장 적합함. (단점이라면 Lua 실행으로 인한 오버헤드이나, 여기선 전혀 문제 없음)

※ 여러 게임에서 사용되는 그 Lua 맞음. 압도적으로 가볍고 임베딩 능력이 뛰어나서 많이 쓰임.

구현하기 in Go — handleEvent

그래서 아래와 같은 Lua 스크립트를 사용하여 Release 트랜잭션을 처리함.

```
const script = `if redis.call("get", KEYS[1]) == ARGV[1] then return redis.call("del", KEYS[1]) else return 0 end`
_, _ = b.redis.Eval(ctx, script, []string{scoreboardLockKey}, token).Result()
```

⇒

```
if redis.call("get", KEYS[1]) == ARGV[1] then
    return redis.call("del", KEYS[1])
 else
    return 0
 end
```

이렇게 해두면 원자적 실행이 보장되니 분산 락 구현을 할 수 있게 됨.

구현하기 in Go — handleEvent

마지막으로 scoreboard.rebuilt 채널에 완료가 되었다는 이벤트를 보내면 handleEvent의 역할은 끝남.

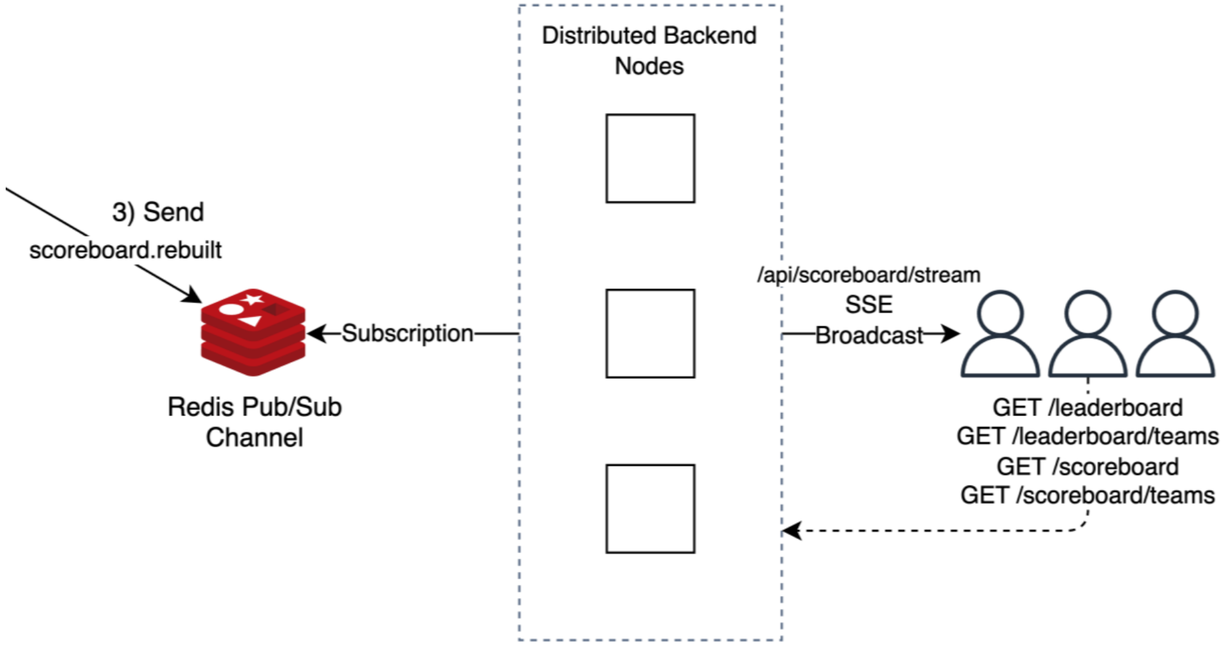
```
_ = b.redis.Publish(ctx, scoreboardRebuiltChannel, payload).Err()
```

여기서도 Publish에 대한 에러 핸들링이 들어갈 수 있으나 여기선 생략함. 오버 엔지니어링임.

구현하기 in Go — scoreboard.rebuilt Subscribe

앞선 handleEvent 처리를 완료했다면 scoreboard.rebuilt Pub/Sub 채널에 이벤트를 발행한다고 하였음.

이는 모든 노드에서 자신의 SSE 스트림에 일관적으로 송신하도록 해야하므로 분산 락과 같은 전략은 필요하지 않음.



클라이언트에서 SSE Connection을 만들고 받는 코드는 생략.

구현하기 in Go — scoreboard.rebuilt Subscribe

초반의 run 함수에서 새로운 Goroutine을 하나 만들고, 해당 채널을 Subscribe 하기만 하면 됨.

```go
go func() {
    ch := rebuilt.Channel()
    for {
        select {
        case <-ctx.Done():
            return
        case msg, ok := <-ch:
            if !ok {
                return
            }

            b.hub.Broadcast(msg.Payload)
        }
    }
}()
```

```go
func (h *SSEHub) Subscribe(buffer int) (chan string, func()) {
    if buffer <= 0 {
        buffer = 1
    }

    ch := make(chan string, buffer)

    h.mu.Lock()
    h.subscribers[ch] = struct{}{}
    h.mu.Unlock()

    unsubscribe := func() {
        h.mu.Lock()
        if _, ok := h.subscribers[ch]; ok {
            delete(h.subscribers, ch)
            close(ch)
        }

        h.mu.Unlock()
    }

    return ch, unsubscribe
}
```

```go
func (h *SSEHub) Broadcast(payload string) {
    h.mu.Lock()
    for ch := range h.subscribers {
        select {
        case ch <- payload:
        default:
        }
    }

    h.mu.Unlock()
}
```

브로드캐스트 로직은 SSE Connections에 대해 같은 메시지를 Fan-out으로 뿌려주기면 하므로 설명은 생략하겠음.

결론

단순히 "기존 스코어보드 API 핸들러의 코드 변경을 최소화하면서 실시간 업데이트 기능 추가"를 시작으로 고안한 주제인데, 여기에 분산 컴퓨팅 및 MSA 구조의 인프라와 백엔드 생태계가 결합되어 더 고민하게 된 주제인 것 같음.

MSA가 표준이 된 생태계에서 분산 처리와 메시징(이벤트) 처리가 가장 핵심적인 구성 요소라고 생각하고, 이에 따라 복잡한 시나리오를 예상, 적절하게 해결하거나 완화하는 것. 설계하는 것이 진정한 실력이지 않나 생각됨.

실제 프로덕션 환경에서 퍼포먼스를 직접 측정해볼 수 있는 기회가 없었기에 스테이징 테스트 / TDD / 유닛 + 통합 테스트 위주로 진행한 것이 다소 아쉬운 부분이지만, 하나의 문제에 대해 트러블 슈팅하면서 여러 개념을 유기적으로 설계하고 구현할 수 있었다는 것에 큰 의의를 두고 있음.

Extra) Redis Pub/Sub 대신 다른 메시징 플랫폼을 사용하는건 안되나요? 당연히 가능.

(Listen/Notify)

등등..

- 다만 **Kafka**는 영속성이 있고 대규모 트래픽 처리에 최적화가 되어있어 너무나도 오버 엔지니어링이고,
- **RabbitMQ**도 마찬가지로 복잡한 큐잉 시스템으로 오버 엔지니어링,
- **PostgreSQL**의 **Listen/Notify**는 DB 기반이기 때문에 처리율 제한, 페이로드 크기 제한 등이 있어 확장성에 부합하지 않다고 판단함.

그리고 무엇보다 기존 Redis를 적극적으로 활용하고 있고, Pub/Sub의 철학과 완벽히 부합하는 구조이며
별도의 구성 없이 바로 사용할 수 있기 때문에 선택하였음.

Thank you for listening .

Github	github.com/**yulmwu**
Discord	@rlawnsdud
Phone	010-2980-1336
Email	• me@swua.kr • normal8781@gmail.com
Linkedin	linkedin.com/in/yulmwu

Reference.

발표를 준비하는 시점에서 필자(발표자)의 일정이 여유롭지 않았으며, 그로 인해 **자료의 완성도가 기대에 미치지 못할 가능성을 충분히 예상**하고 있었습니다. 이러한 이유로 한때는 널카말카 세미나에 참여하지 않는 방안도 고려했었습니다.

그러나 오랜 전통과 가치를 이어온 널카말카 세미나에 직접 참여하지 않는 것은, 평소 모든 동아리 부원들에게 **"의미 있는 활동인 만큼 반드시 참여해야 한다"**는 취지의 이야기를 해왔던 입장과도 **배치되는 행동**이라 판단하였습니다.
스스로에게 요구하는 기준과 구성원들에게 권고하는 기준이 달라져서는 안 된다는 점에서,
이를 외면하는 것은 모순이라 생각하였습니다.

비록 발표 자료의 완성도가 다소 아쉬울 수는 있을지라도, 보안과 전공 동아리의 중대한 행사에는 책임감을 가지고 참여하는 것이 마땅한 자세라고 판단하여 본 발표를 준비하게 되었습니다.

아울러 본 발표 자료는 **2026년 2월 말**에 작성된 내용을 기반으로 하고 있습니다. 따라서 이후의 기술적 변화나 최신 동향이 일부 반영되지 않았을 수 있으니, 이 점을 감안하여 참고해 주시기 바랍니다.