DEVIEW
2019

ms 단위의 Serverless World에서 Docker의 성능 한계 극복하기

김동경
PaaS

NAVER

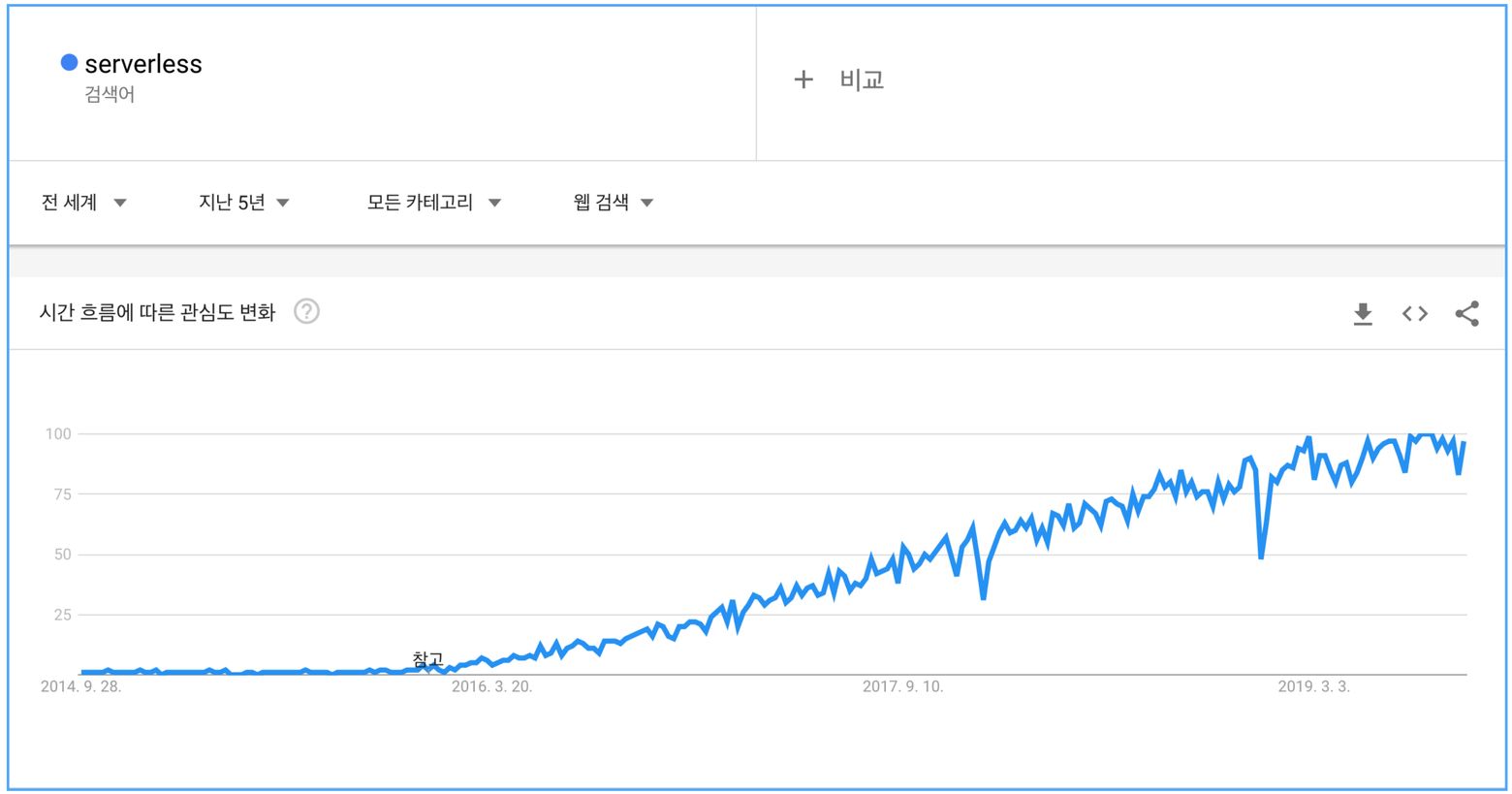
CONTENTS

DEVIEW 2019

Serverless Computing?

Trends

DEVIEW 2019



Trends

NAVER

Trends

APACHE OpenWhisk™

Kubeless

Knative

현대적인 서버리스 워크로드를 빌드, 배포, 관리할 수 있는 Kubernetes 기반 플랫폼입니다.

자세히 알아보기 Cloud Run 사용해 보기

IronFunctions

OPENFAAS

Fn Project

Open Source. Container-native. Serverless platform.

fission

NAVER

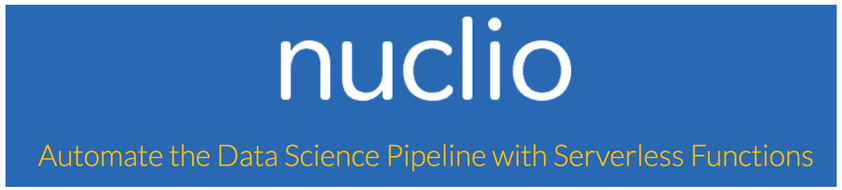
Trends

인증

앱 관리

보안

모니터링



Data Science

NAVER

Trends

Serverless Computing: The Next Cloud Evolution

EGT BLOG

Serverless explainer: The next generation of cloud infrastructure

Serverless computing is a new way of hosting applications on infrastructure that end users do not manage

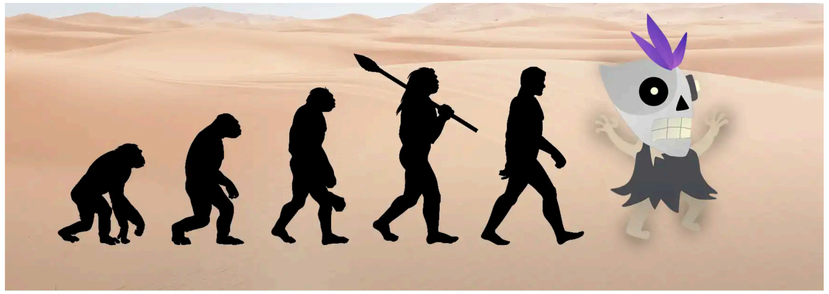
By Brandon Butler

Senior Editor, Network World | APR 3, 2017 11:00 AM PDT

Why Serverless is the future of cloud computing?

Surya Jakhotia Follow

May 26, 2018 · 4 min read ★

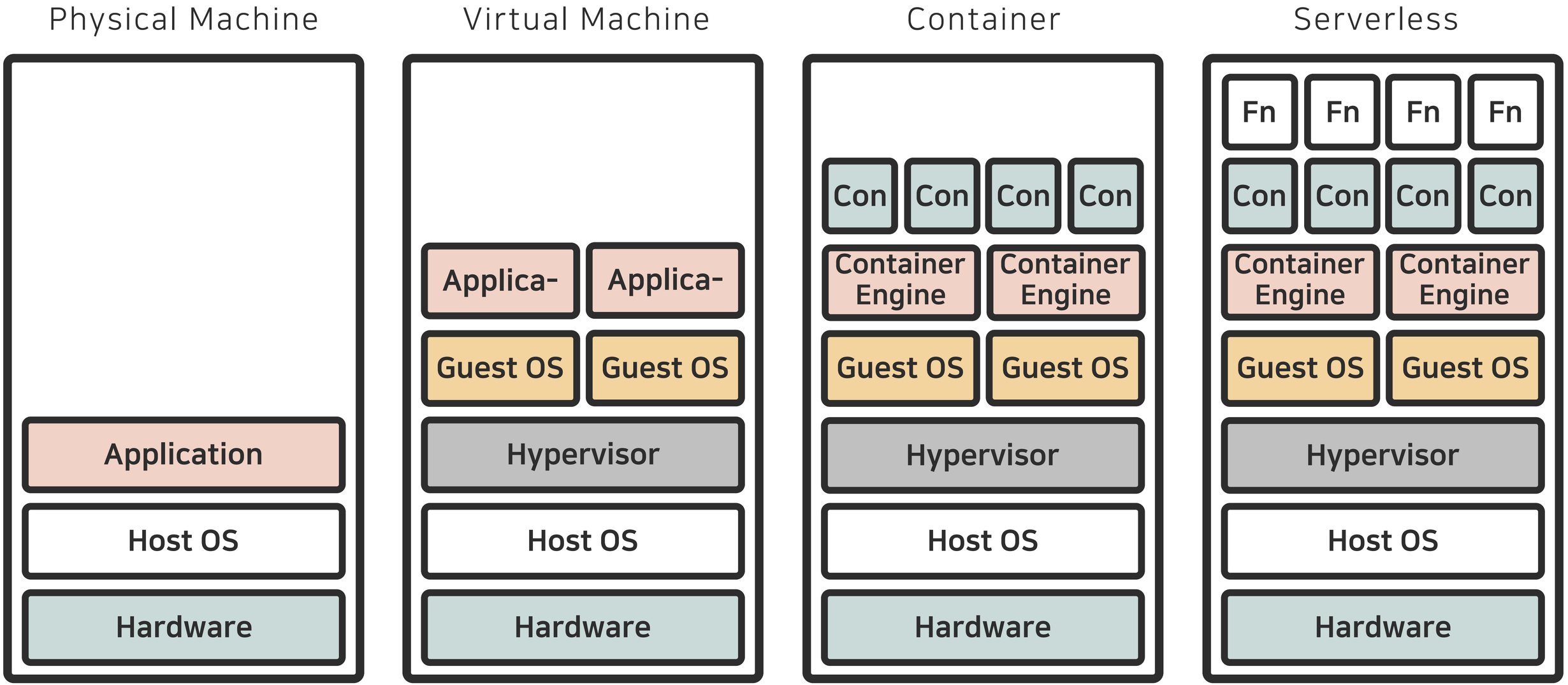


The Road to Serverless: The Evolution of Cloud Computing

The next logical step in the advancement and evolutionary progression of cloud computing is serverless.

NAVER

Cloud의 진화 과정



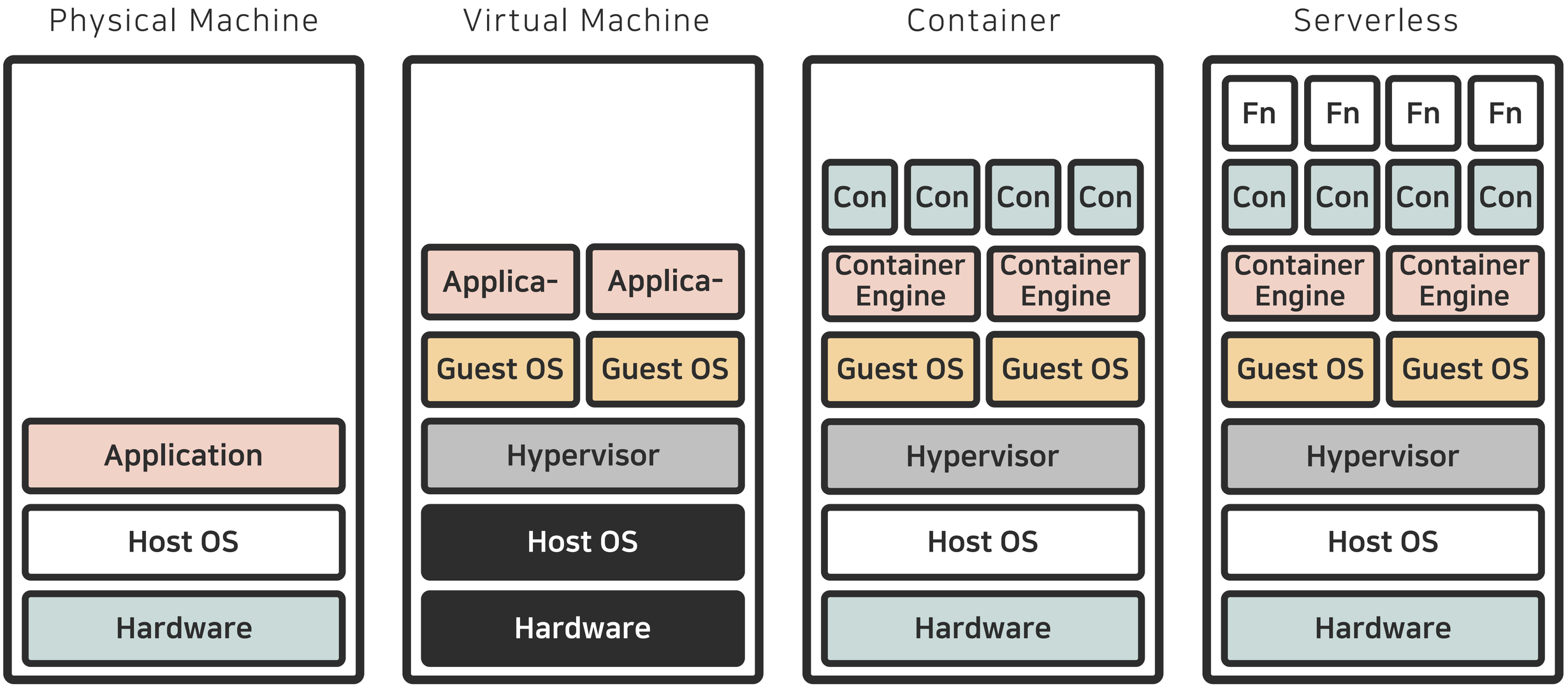
하드웨어 독점 사용

하드웨어 리소스 공유

VM 리소스 공유

시간축으로 리소스 공유

Cloud의 진화 과정



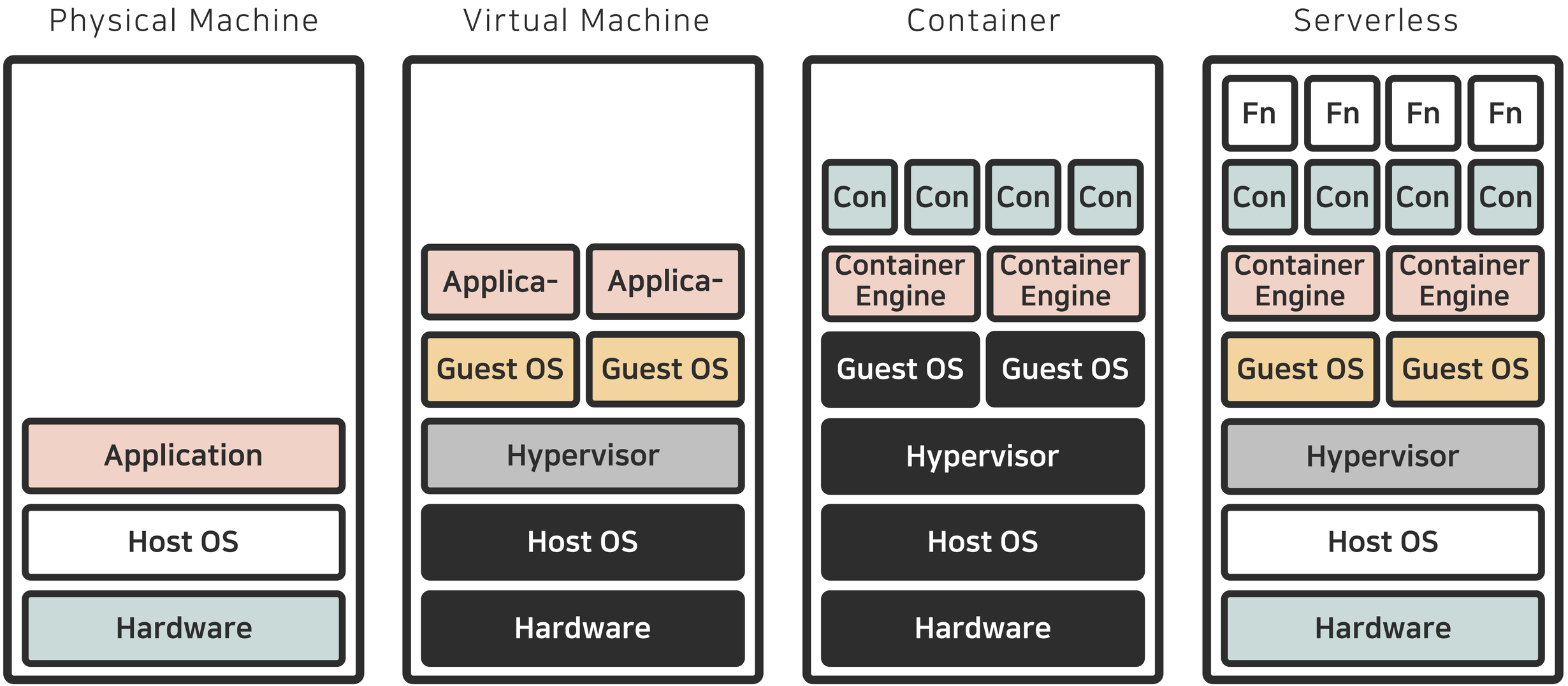
하드웨어 독점 사용

하드웨어 리소스 공유

VM 리소스 공유

시간축으로 리소스 공유

Cloud의 진화 과정



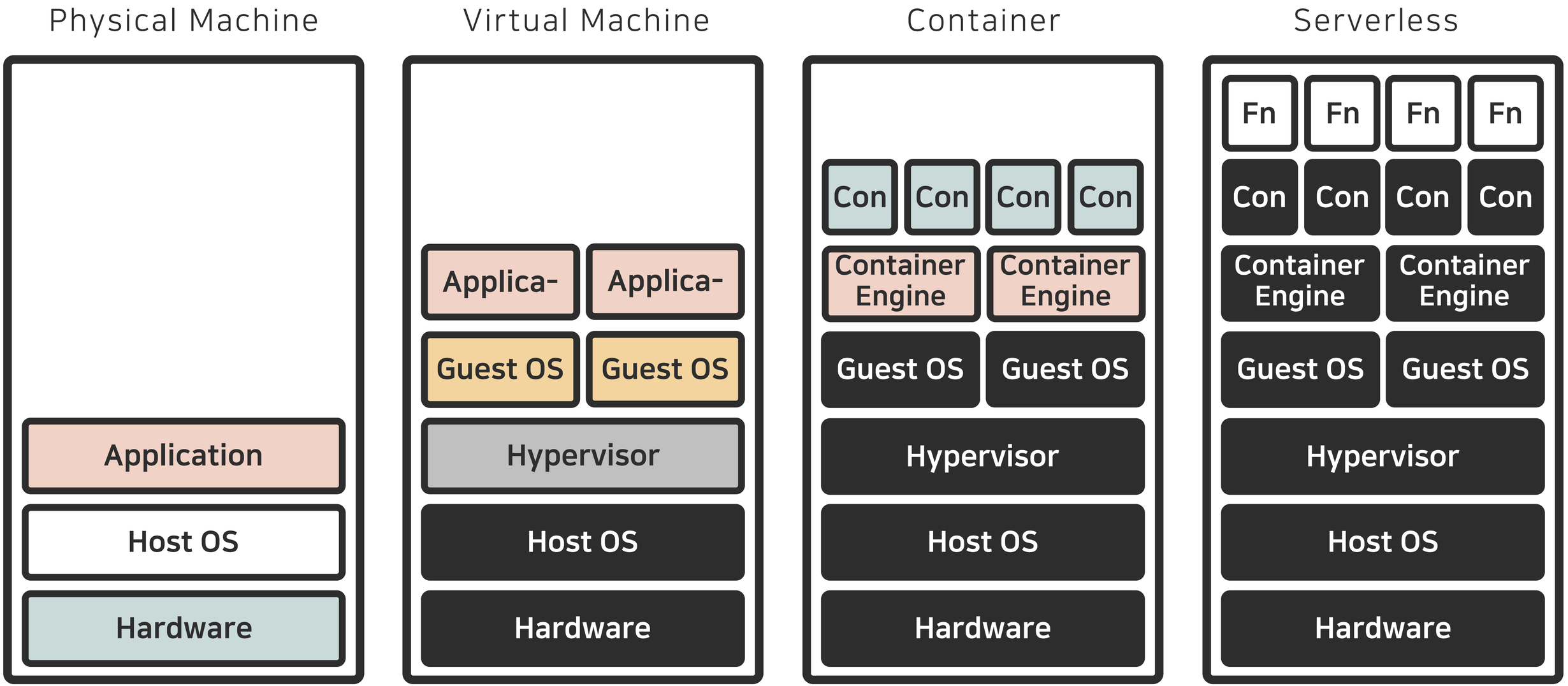
하드웨어 독점 사용

하드웨어 리소스 공유

VM 리소스 공유

시간축으로 리소스 공유

Cloud의 진화 과정



하드웨어 독점 사용

하드웨어 리소스 공유

VM 리소스 공유

시간축으로 리소스 공유

Serverless의 종류

DEVIEW 2019

AWS 서버리스 플랫폼

AWS에서는 서버리스 애플리케이션을 구축 및 실행하는 데 사용할 수 있는 일련의 완전 관리형 서비스를 제공합니다. 서버리스 애플리케이션은 컴퓨팅, 데이터베이스, 스토리지, 스트림 처리, 메시지 대기열 등의 백엔드 구성 요소를 위한 서버를 프로비저닝, 유지 및 관리할 필요가 없습니다. 그뿐만 아니라 더는 애플리케이션 내결함성 및 가용성에 관해 걱정할 필요가 없습니다. AWS가 이러한 모든 기능을 대신 처리합니다. 이를 통해 제품 출시 시간을 단축하면서 제품 혁신에 집중할 수 있습니다.

컴퓨팅

AWS Lambda를 사용하면 서버를 프로비저닝하거나 관리할 필요 없이 코드를 실행할 수 있습니다. 사용한 컴퓨팅 시간만큼만 비용을 지불하고, 코드가 실행되지 않을 때는 요금이 부과되지 않습니다.

Lambda@Edge를 사용하면 Amazon CloudFront 이벤트에 대한 응답으로 AWS 엣지 로케이션에서 Lambda 함수를 실행할 수 있습니다.

AWS Fargate는 컨테이너용으로 특별히 빌드된 서버리스 컴퓨팅 엔진입니다. Fargate는 컨테이너 실행에 필요한 인프라를 조정 및 관리합니다.

스토리지

Amazon Simple Storage Service(Amazon S3)는 개발자와 IT 팀에 안전하고 안정성과 확장성이 뛰어난 객체 스토리지를 제공합니다. Amazon S3는 간단한 웹 서비스 인터페이스를 통해 웹 어디서나 원하는 양의 데이터를 저장 및 검색할 수 있으므로 사용하기가 쉽습니다.

Amazon Elastic File System(EFS)은 간단하고 확장 가능하며 탄력적인 파일 스토리지를 제공합니다. 수요에 따라 탄력적으로 크기를 조정하여 파일을 추가 또는 제거할 때 자동적으로 확장 또는 축소되도록 빌드되었습니다.

데이터 스토어

Amazon DynamoDB는 규모에 관계없이 10밀리초 미만의 지연 시간이 일관되게 요구되는 모든 애플리케이션을 위한 빠르고 유연성이 뛰어난 NoSQL 데이터베이스 서비스입니다.

Amazon Aurora 서버리스는 Amazon Aurora(MySQL 호환 에디션)를 위한 온디맨드 Auto Scaling 구성입니다. 이를 통해 데이터베이스를 자동으로 시작 및 종료하고, 애플리케이션의 필요에 따라 용량을 늘리거나 줄일 수 있습니다.

API 프록시

Amazon API Gateway는 어떤 규모에서든 개발자가 API를 손쉽게 생성, 게시, 유지 관리, 모니터링 및 보호할 수 있도록 지원하는 완전관리형 서비스입니다. 이 서비스에서는 API 관리를 위한 포괄적인 플랫폼을 제공합니다. API Gateway를 통해 수십만 개의 동시 API 호출을 처리하고 트래픽 관리, 권한 부여 및 액세스 제어, 모니터링 및 API 버전 관리를 처리할 수 있습니다.

애플리케이션 통합

Amazon SNS는 마이크로서비스, 분산 시스템 및 서버리스 애플리케이션을 쉽게 분리하고 확장할 수 있게 해주는 완전관리형 게시/구독 메시징 서비스입니다.

오케스트레이션

AWS Step Functions를 사용하면 시각적 워크플로를 사용해 분산 애플리케이션 및 마이크로서비스의 구성 요소를 손쉽게 조정할 수 있습니다. 각각 기능을 수행하는 개별 구성 요

분석

Amazon Kinesis는 AWS의 스트리밍 데이터를 위한 플랫폼으로서, 스트리밍 데이터를 손쉽게 로드 및 분석할 수 있는 강력한 서비스를 제공하고, 특정 요구에 맞게 사용자 지정 스트

개발자 도구

AWS에서는 개발자가 서버리스 애플리케이션 개발 프로세스에 사용할 수 있는 도구와 서비스를 제공합니다. AWS와 AWS 파트너 에코시스템은 지속적 통합 및 전달, 테스트, 배포,

Backend as a Service

Database as a Service

Function as a Service

Serverless의 종류

DEVIEW 2019

AWS 서버리스 플랫폼

AWS에서는 서버리스 애플리케이션을 구축 및 실행하는 데 사용할 수 있는 일련의 완전 관리형 서비스를 제공합니다. 서버리스 애플리케이션은 컴퓨팅, 데이터베이스, 스토리지, 스트림 처리, 메시지 대기열 등의 백엔드 구성 요소를 위한 서버를 프로비저닝, 유지 및 관리할 필요가 없습니다. 그뿐만 아니라 더는 애플리케이션 내결함성 및 가용성에 관해 걱정할 필요가 없습니다. AWS가 이러한 모든 기능을 대신 처리합니다. 이를 통해 제품 출시 시간을 단축하면서 제품 혁신에 집중할 수 있습니다.

컴퓨팅

AWS Lambda를 사용하면 서버를 프로비저닝하거나 관리할 필요 없이 코드를 실행할 수 있습니다. 사용한 컴퓨팅 시간만큼만 비용을 지불하고, 코드가 실행되지 않을 때는 요금이 부과되지 않습니다.

Lambda@Edge를 사용하면 Amazon CloudFront 이벤트에 대한 응답으로 AWS 엣지 로케이션에서 Lambda 함수를 실행할 수 있습니다.

AWS Fargate는 컨테이너용으로 특별히 빌드된 서버리스 컴퓨팅 엔진입니다. Fargate는 컨테이너 실행에 필요한 인프라를 조정 및 관리합니다.

스토리지

Amazon Simple Storage Service(Amazon S3)는 개발자와 IT 팀에 안전하고 안정성과 확장성이 뛰어난 객체 스토리지를 제공합니다. Amazon S3는 간단한 웹 서비스 인터페이스를 통해 웹 어디서나 원하는 양의 데이터를 저장 및 검색할 수 있으므로 사용하기가 쉽습니다.

Amazon Elastic File System(EFS)은 간단하고 확장 가능하며 탄력적인 파일 스토리지를 제공합니다. 수요에 따라 탄력적으로 크기를 조정하여 파일을 추가 또는 제거할 때 자동적으로 확장 또는 축소되도록 빌드되었습니다.

데이터 스토어

Amazon DynamoDB는 규모에 관계없이 10밀리초 미만의 지연 시간이 일관되게 요구되는 모든 애플리케이션을 위한 빠르고 유연성이 뛰어난 NoSQL 데이터베이스 서비스입니다.

Amazon Aurora 서버리스는 Amazon Aurora(MySQL 호환 에디션)를 위한 온디맨드 Auto Scaling 구성입니다. 이를 통해 데이터베이스를 자동으로 시작 및 종료하고, 애플리케이션의 필요에 따라 용량을 늘리거나 줄일 수 있습니다.

API 프록시

Amazon API Gateway는 어떤 규모에서든 개발자가 API를 손쉽게 생성, 게시, 유지 관리, 모니터링 및 보호할 수 있도록 지원하는 완전관리형 서비스입니다. 이 서비스에서는 API 관리를 위한 포괄적인 플랫폼을 제공합니다. API Gateway를 통해 수십만 개의 동시 API 호출을 처리하고 트래픽 관리, 권한 부여 및 액세스 제어, 모니터링 및 API 버전 관리를 처리할 수 있습니다.

애플리케이션 통합

Amazon SNS는 마이크로서비스, 분산 시스템 및 서버리스 애플리케이션을 쉽게 분리하고 확장할 수 있게 해주는 완전관리형 게시/구독 메시징 서비스입니다.

오케스트레이션

AWS Step Functions를 사용하면 시각적 워크플로를 사용해 분산 애플리케이션 및 마이크로서비스의 구성 요소를 손쉽게 조정할 수 있습니다. 각각 기능을 수행하는 개별 구성 요

분석

Amazon Kinesis는 AWS의 스트리밍 데이터를 위한 플랫폼으로서, 스트리밍 데이터를 손쉽게 로드 및 분석할 수 있는 강력한 서비스를 제공하고, 특정 요구에 맞게 사용자 지정 스트

개발자 도구

AWS에서는 개발자가 서버리스 애플리케이션 개발 프로세스에 사용할 수 있는 도구와 서비스를 제공합니다. AWS와 AWS 파트너 에코시스템은 지속적 통합 및 전달, 테스트, 배포,

Backend as a Service

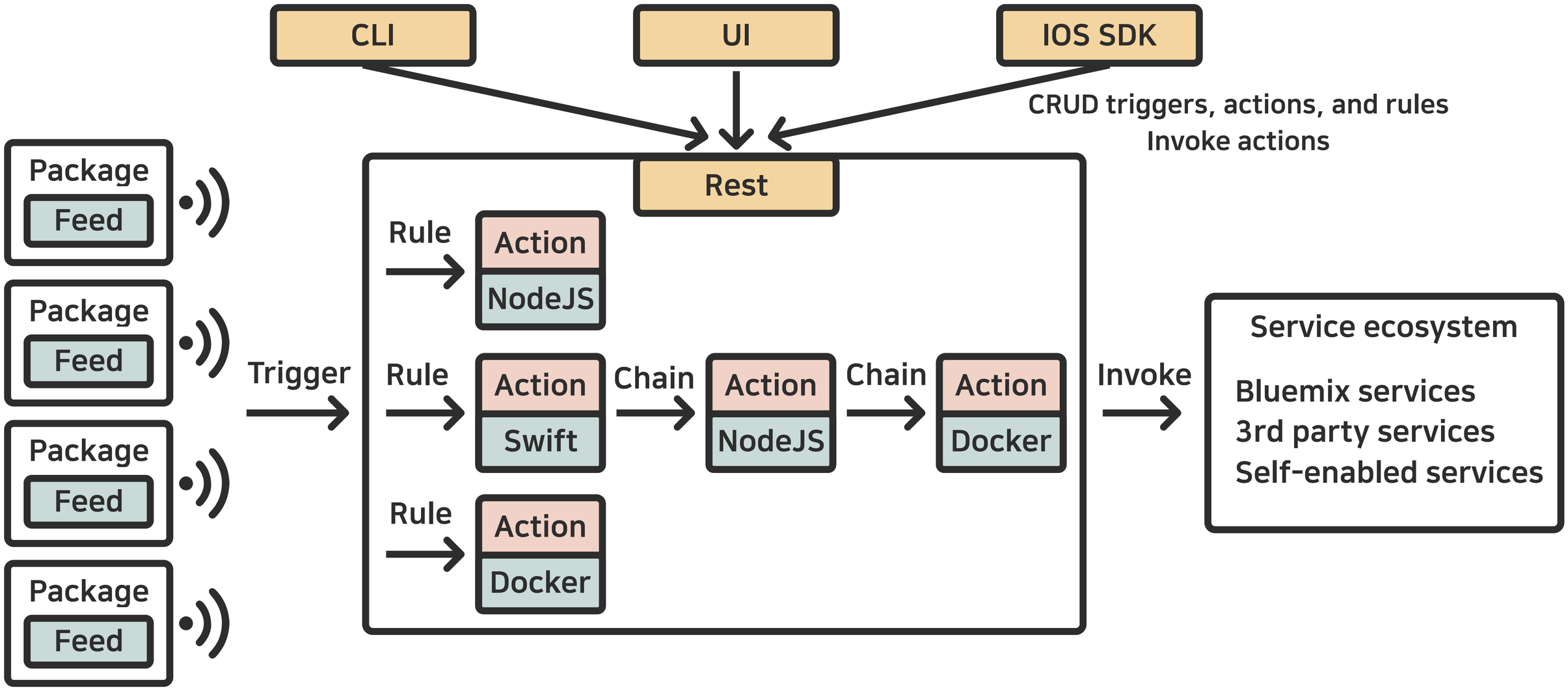
Database as a Service

Function as a Service

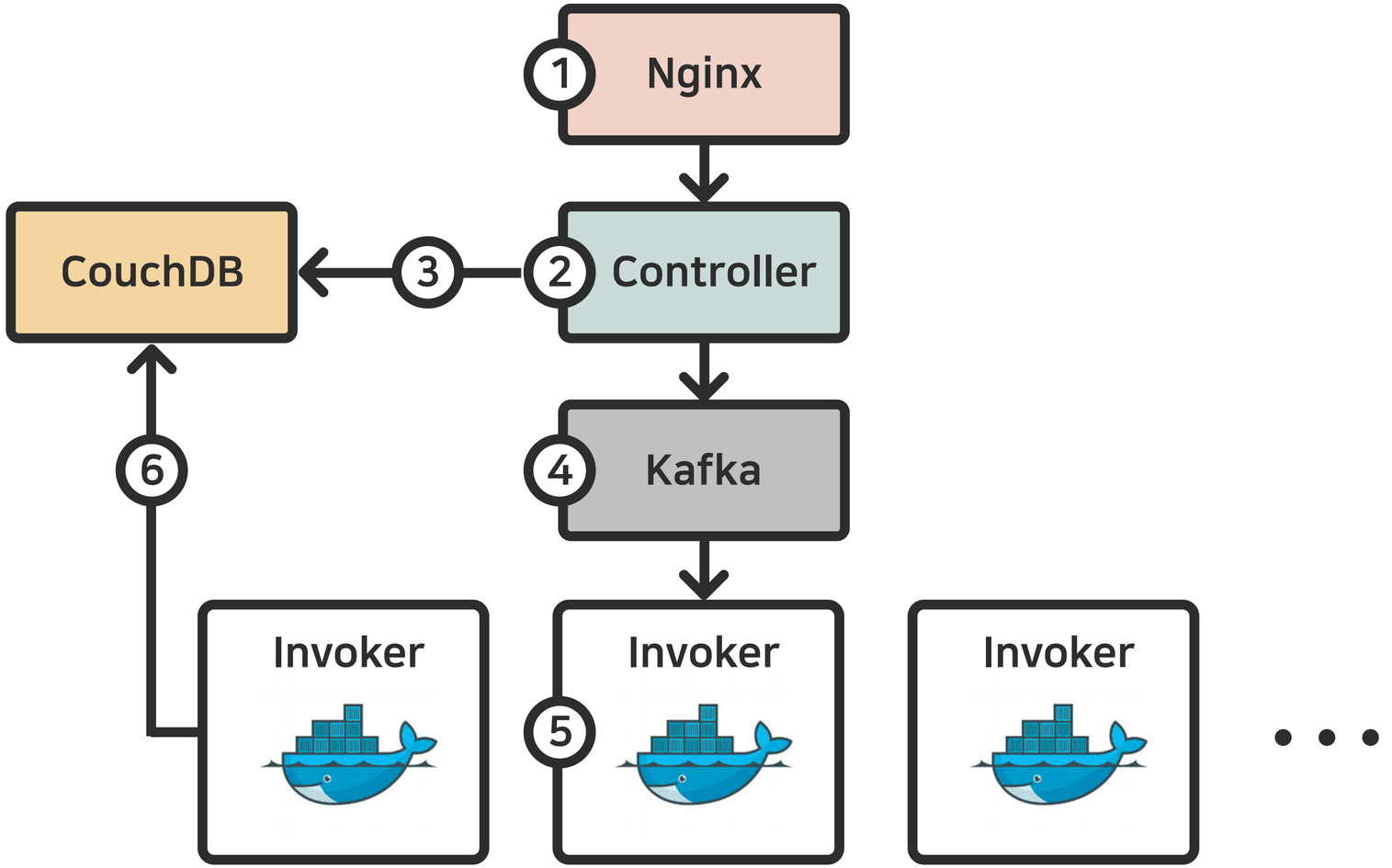
What is OpenWhisk?

What is Apache OpenWhisk?

DEVIEW 2019



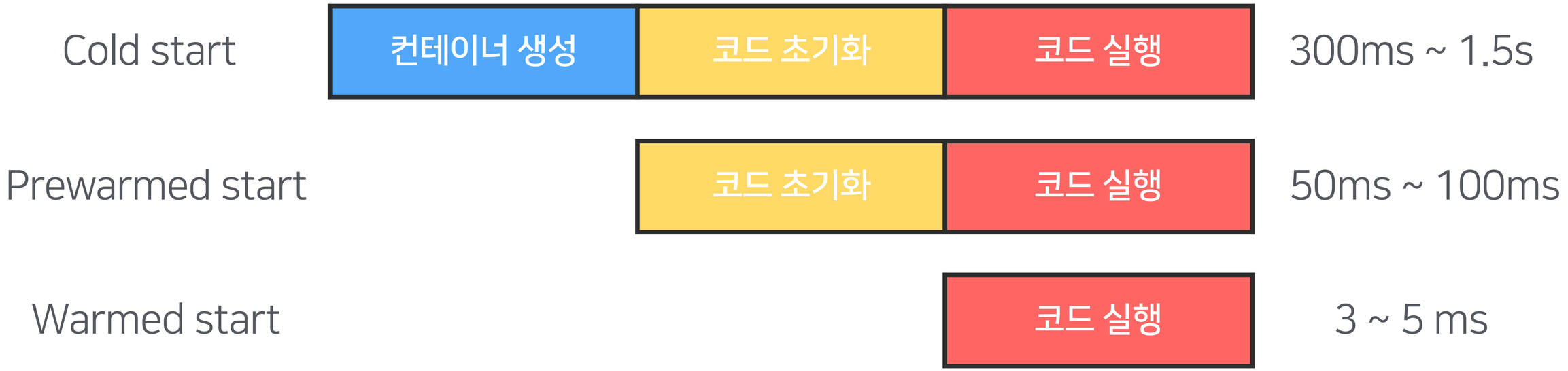
How to work?


1
Nginx
2
Controller
3
CouchDB
4
Kafka
5
6
Invoker
Invoker
Invoker
• • •

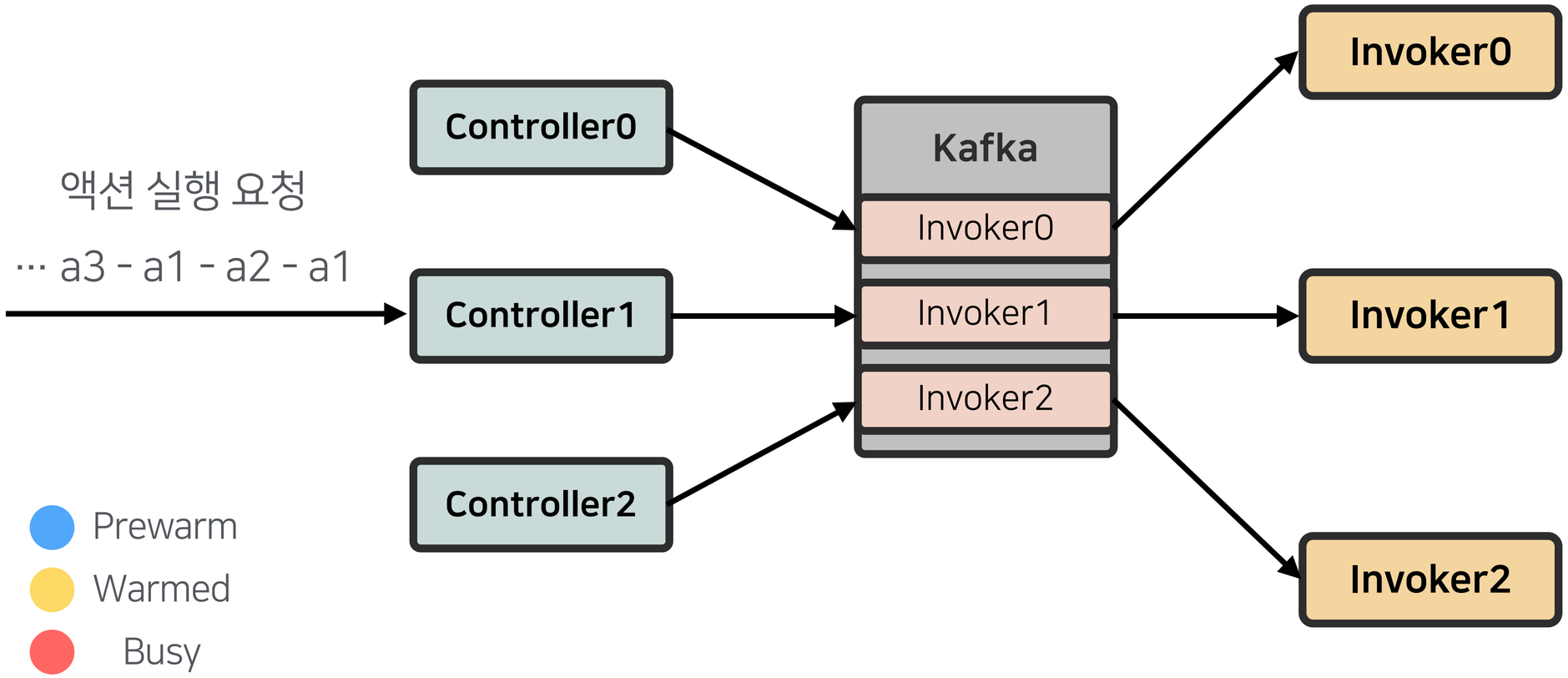
액션의 시작 형태

DEVIEW
2019

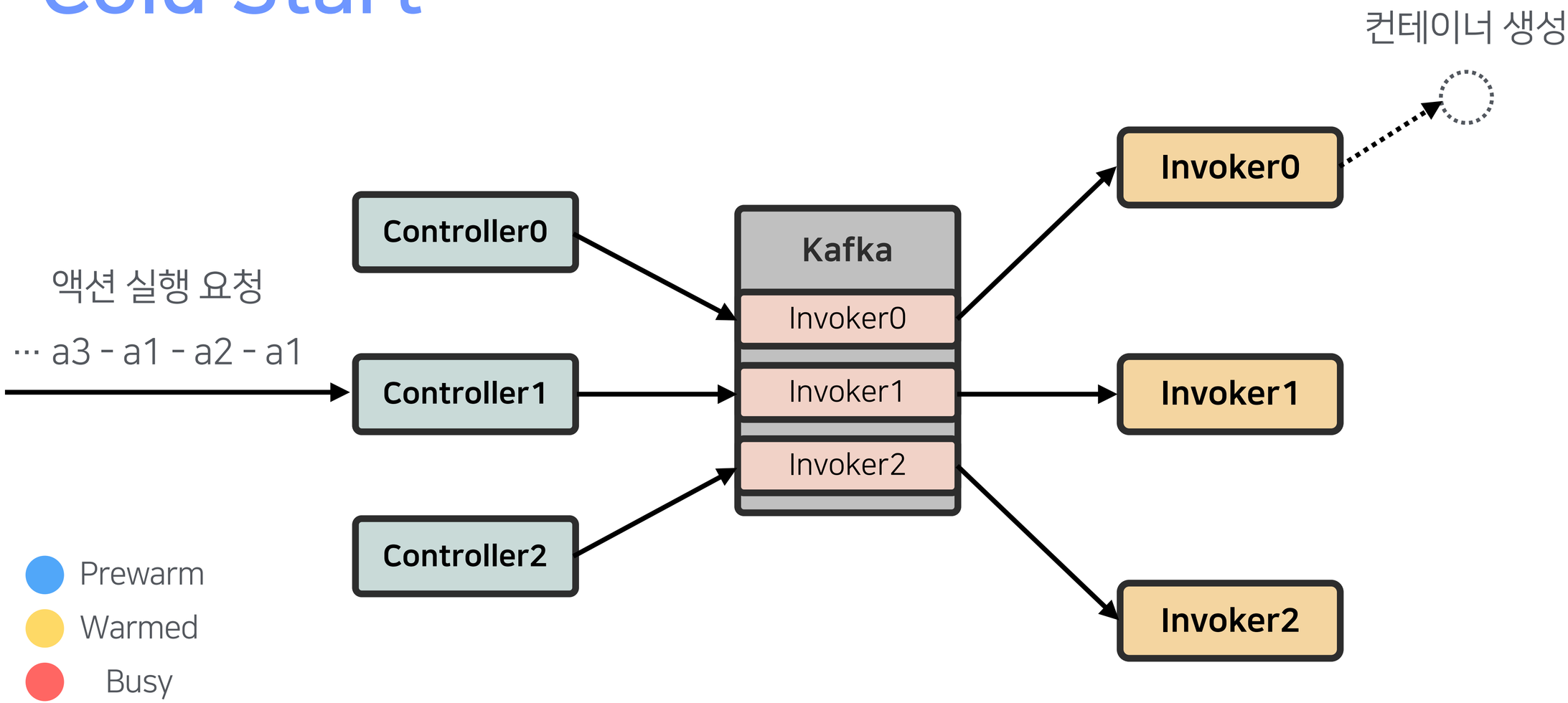
액션의 시작 형태


Cold start
컨테이너 생성
코드 초기화
코드 실행
300ms ~ 1.5s
Prewarmed start
코드 초기화
코드 실행
50ms ~ 100ms
Warmed start
코드 실행
3 ~ 5 ms

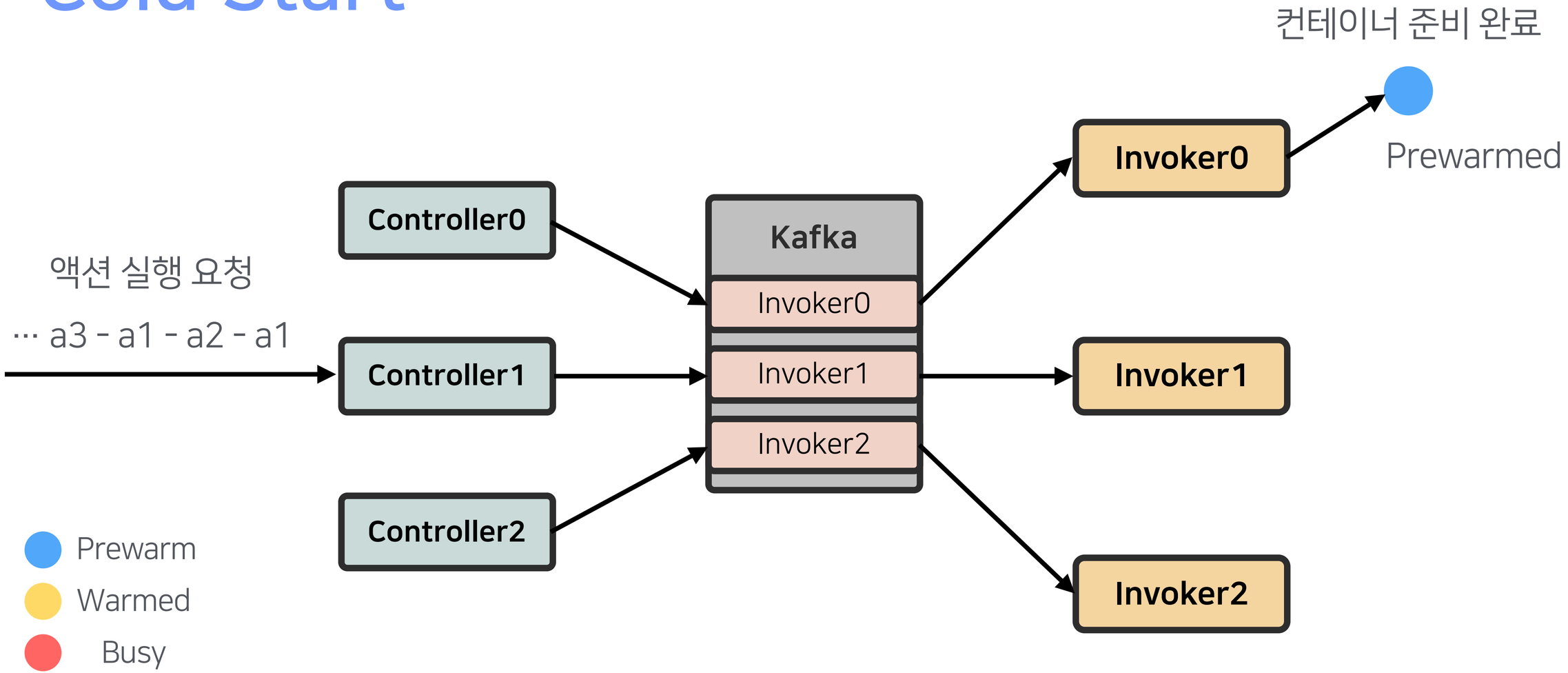
Cold Start


액션 실행 요청
··· a3 - a1 - a2 - a1
Controller0
Controller1
Controller2
Kafka
Invoker0
Invoker1
Invoker2
Invoker0
Invoker1
Invoker2
Prewarm
Warmed
Busy

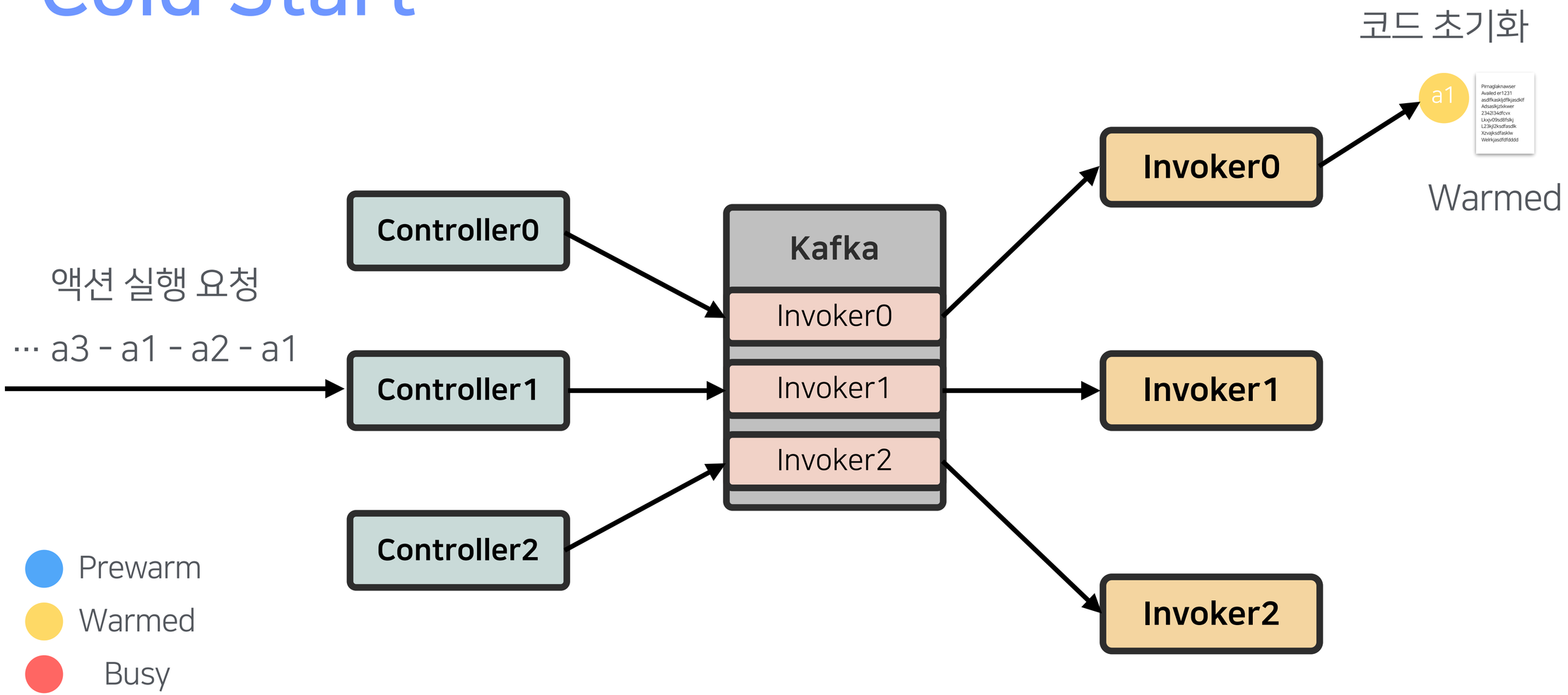
Cold Start


컨테이너 생성
Invoker0
Controller0
Kafka
액션 실행 요청
Invoker0
··· a3 - a1 - a2 - a1
Controller1
Invoker1
Invoker1
Invoker2
Controller2
Invoker2
Prewarm
Warmed
Busy

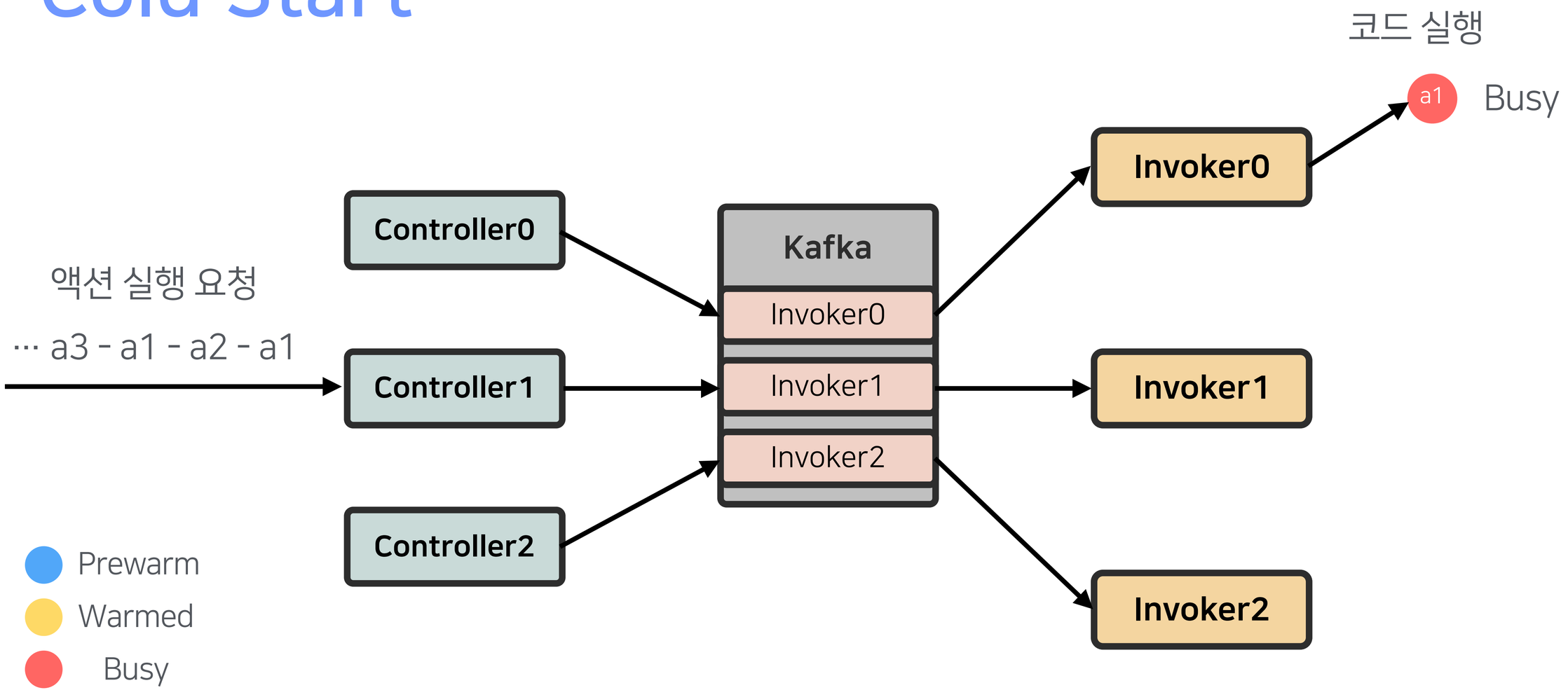
Cold Start


컨테이너 준비 완료
Prewarmed
Invoker0
액션 실행 요청
··· a3 - a1 - a2 - a1
Controller0
Controller1
Controller2
Kafka
Invoker0
Invoker1
Invoker2
Invoker0
Invoker1
Invoker2
Prewarm
Warmed
Busy

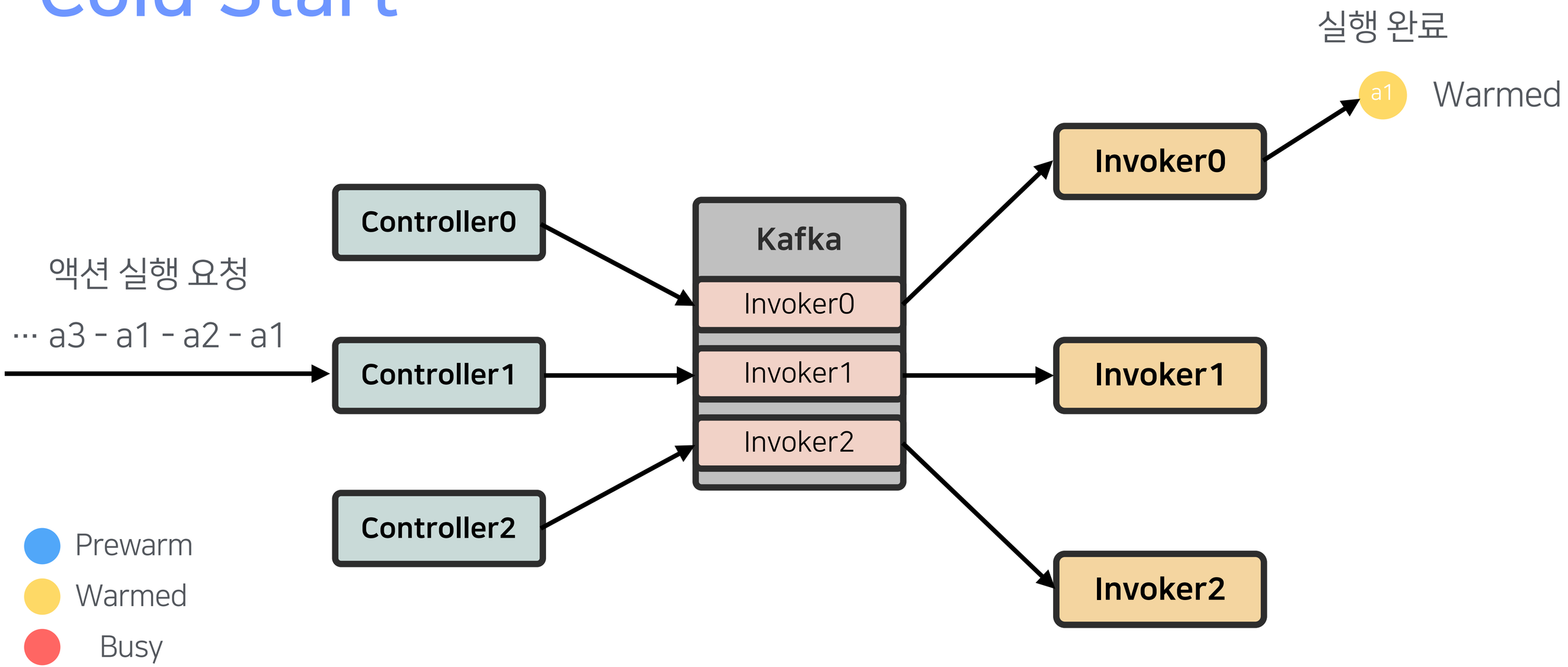
Cold Start


코드 초기화
a1
Warmed
액션 실행 요청
… a3 - a1 - a2 - a1
Controller0
Controller1
Controller2
Kafka
Invoker0
Invoker1
Invoker2
Invoker0
Invoker1
Invoker2
Prewarm
Warmed
Busy

Cold Start

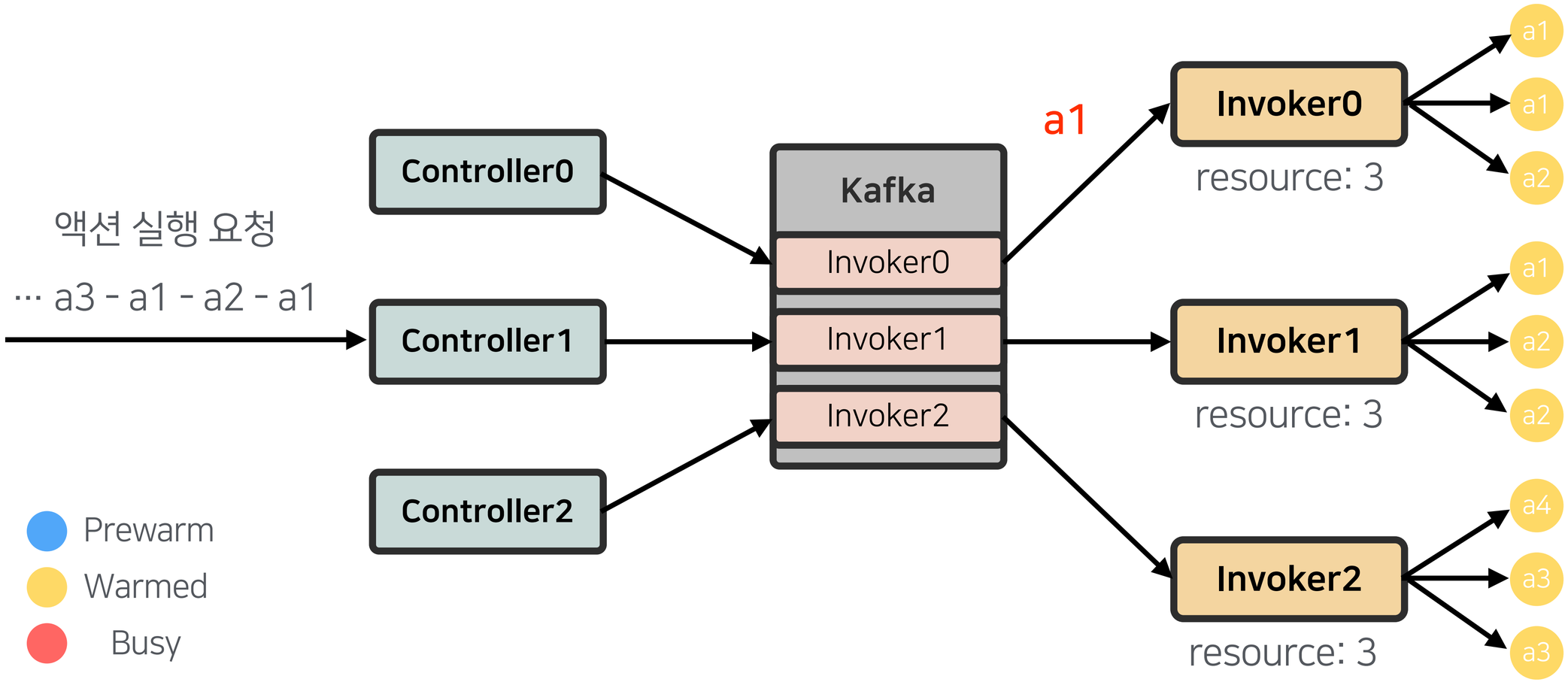

코드 실행
a1
Busy
Invoker0
액션 실행 요청
··· a3 - a1 - a2 - a1
Controller0
Kafka
Invoker0
Controller1
Invoker1
Invoker1
Invoker2
Controller2
Invoker2
Prewarm
Warmed
Busy

Cold Start

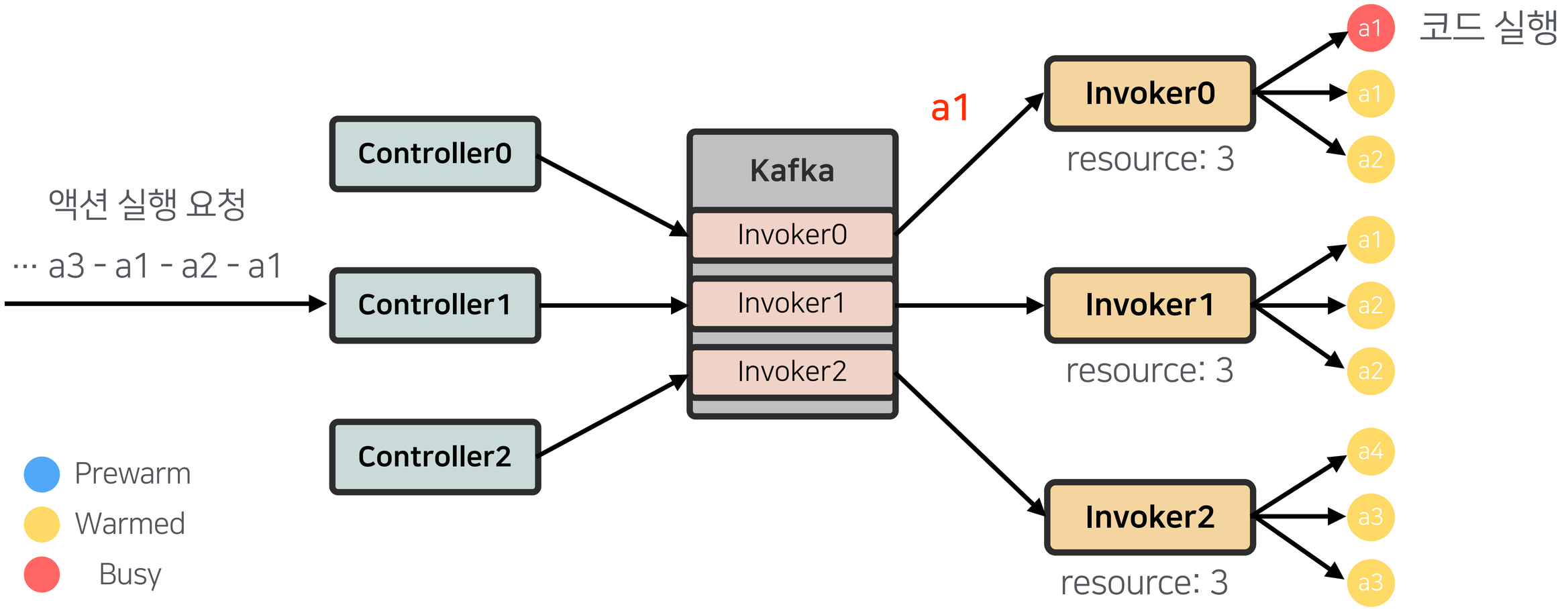

실행 완료
a1
Warmed
Invoker0
Controller0
Kafka
액션 실행 요청
… a3 - a1 - a2 - a1
Invoker0
Invoker1
Controller1
Invoker1
Invoker2
Controller2
Invoker2
Prewarm
Warmed
Busy

액션의 실행

DEVIEW 2019

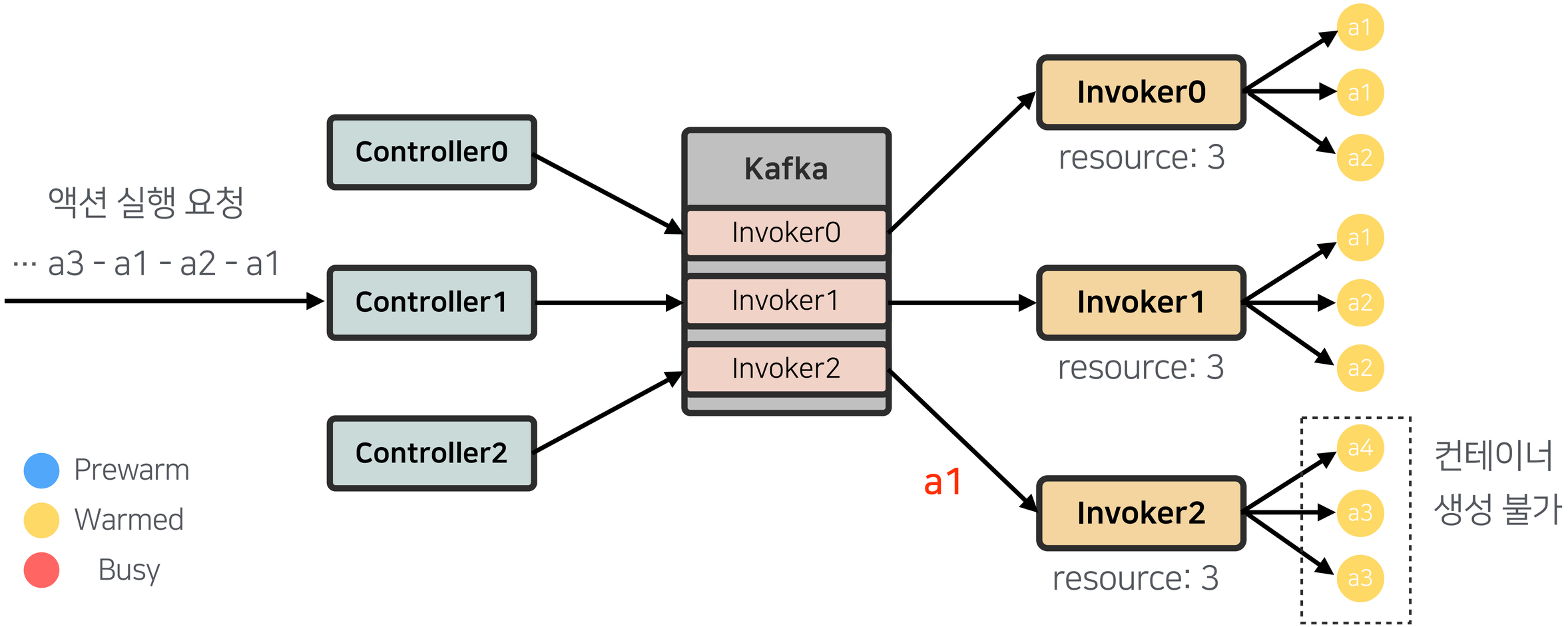


액션의 실행


액션 실행 요청
··· a3 - a1 - a2 - a1
Controller0
Controller1
Controller2
Kafka
Invoker0
Invoker1
Invoker2
a1
Invoker0
resource: 3
Invoker1
resource: 3
Invoker2
resource: 3
코드 실행
a1
a1
a2
a1
a2
a2
a4
a3
a3
Prewarm
Warmed
Busy

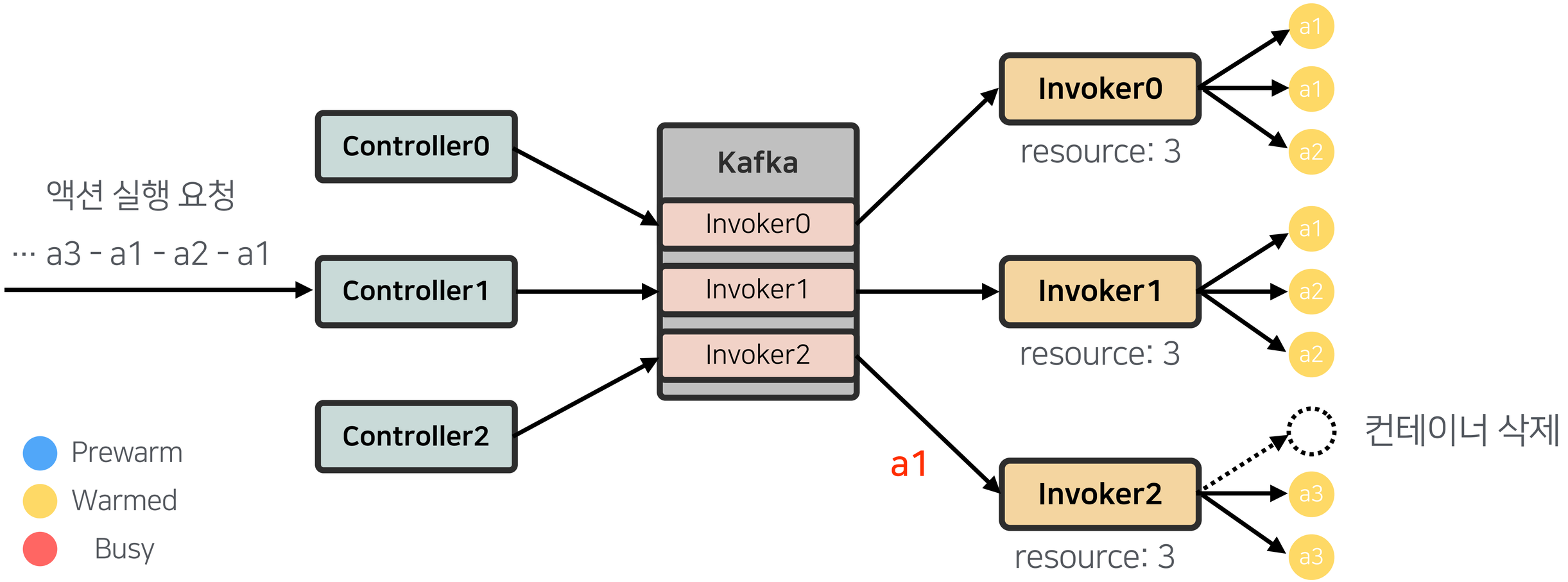
액션의 실행

DEVIEW 2019



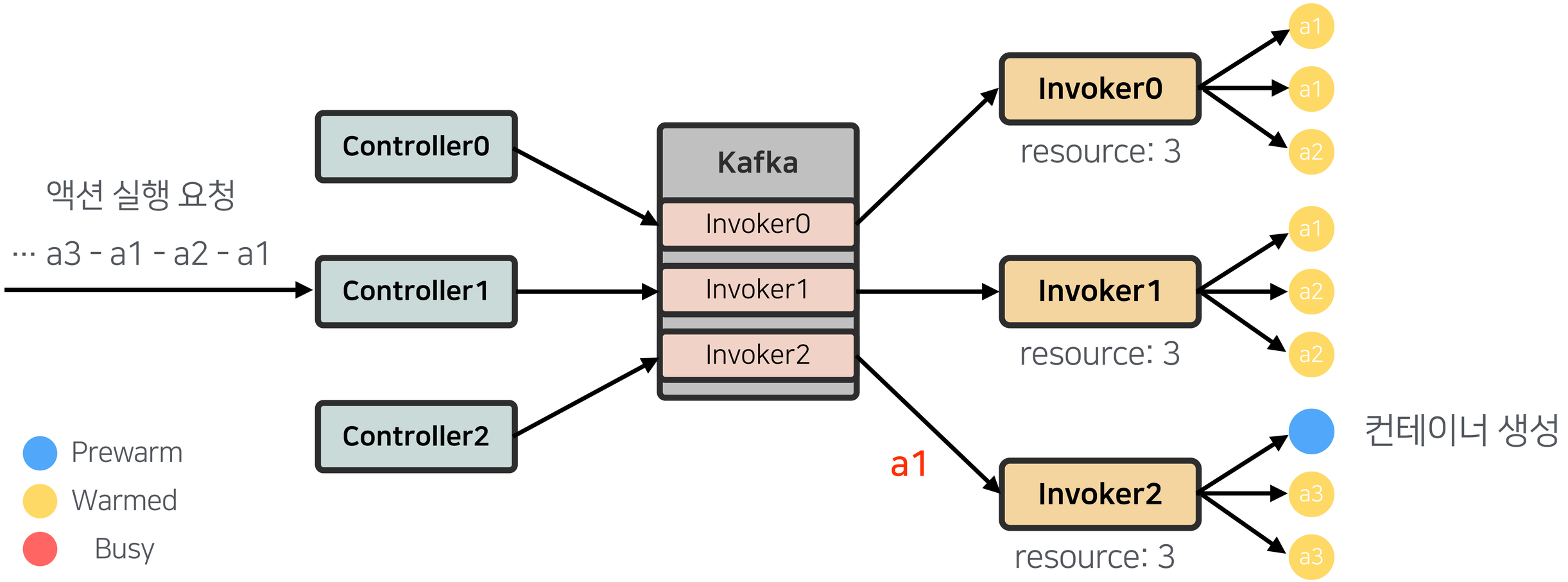
액션의 실행

DEVIEW 2019



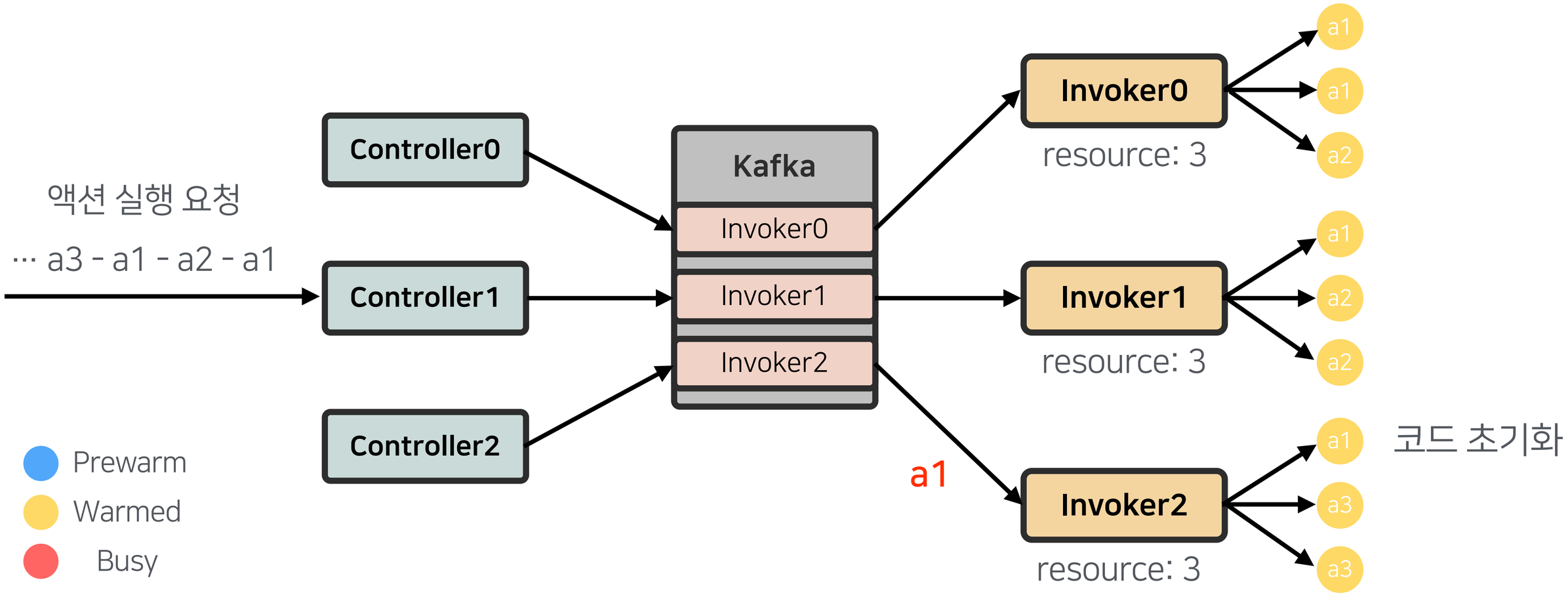
액션의 실행

DEVIEW 2019

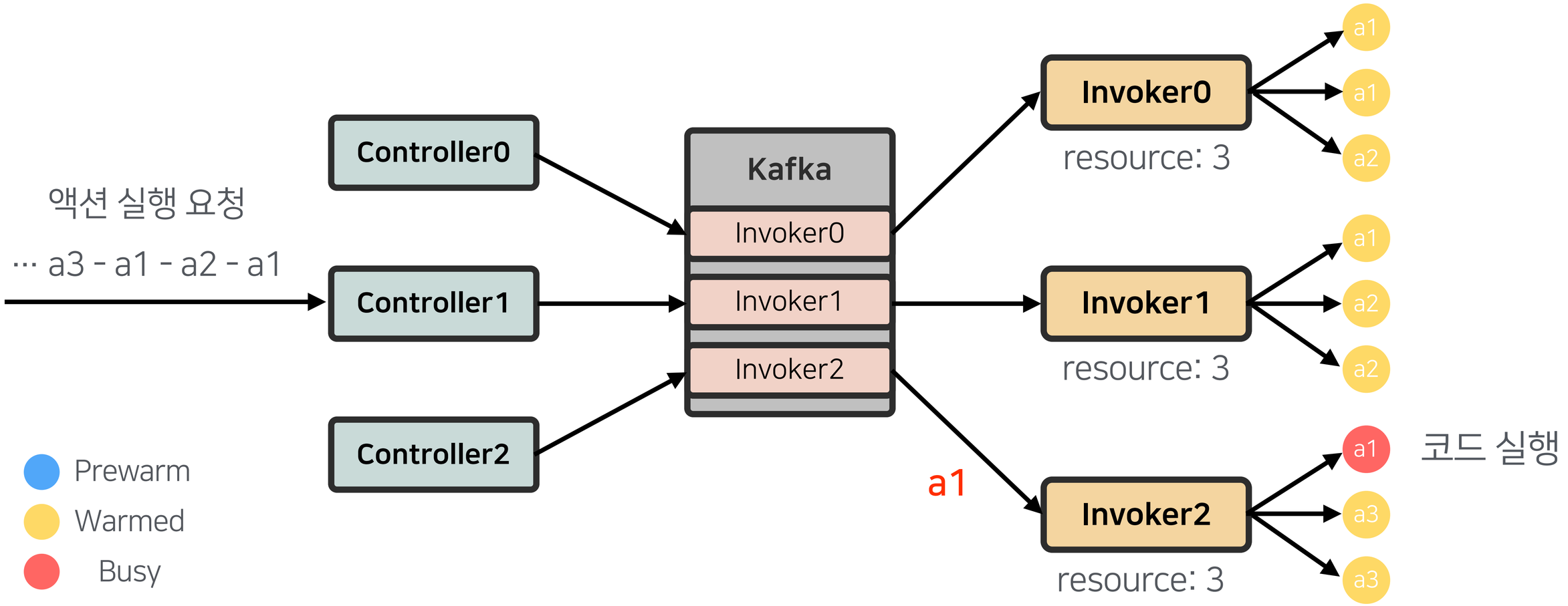


액션의 실행

DEVIEW 2019



액션의 실행


액션 실행 요청
··· a3 - a1 - a2 - a1
Controller0
Controller1
Controller2
Kafka
Invoker0
Invoker1
Invoker2
Invoker0
resource: 3
Invoker1
resource: 3
Invoker2
resource: 3
a1
a1
a2
a1
a2
a2
a1
a3
a3
a1
코드 실행
Prewarm
Warmed
Busy

OpenWhisk에서 스케쥴링의 의미

액션 실행 요청을

어떤 Invoker에게 보낼지 결정

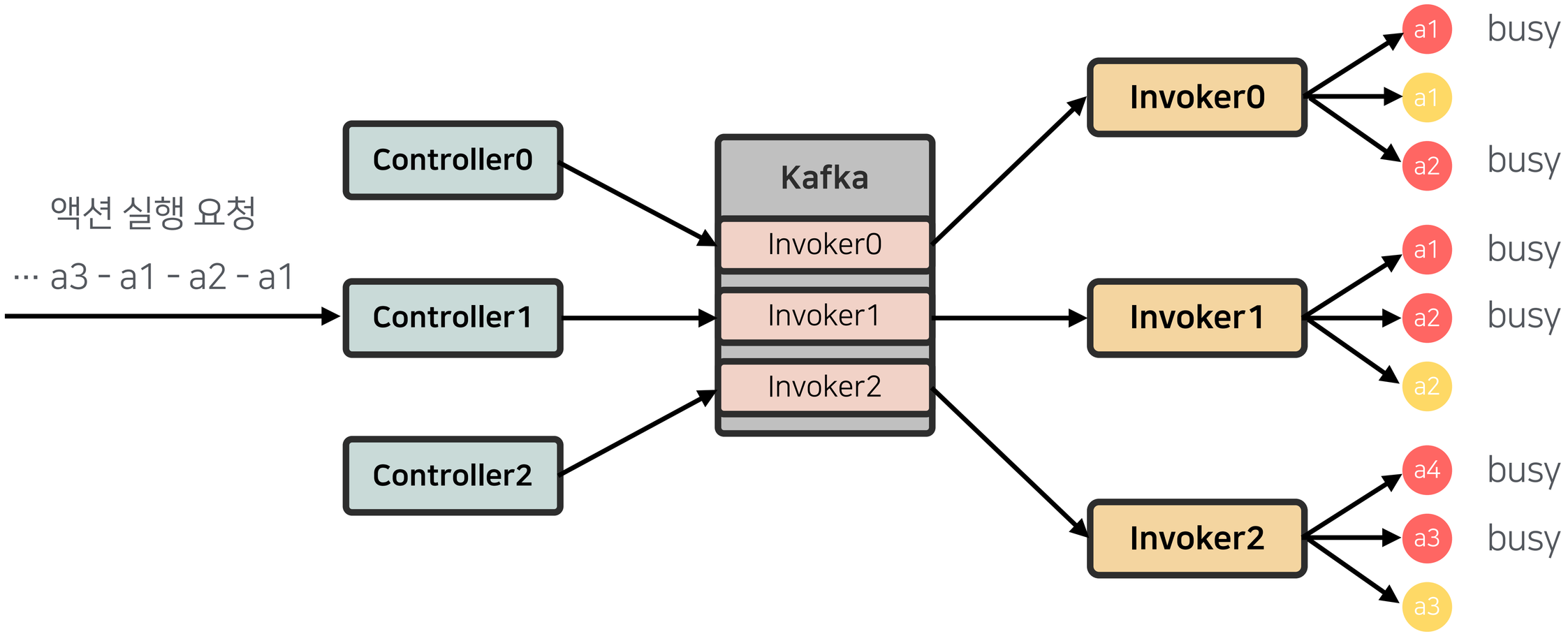
OpenWhisk에서 스케쥴링의 의미

이미 구동되어 있는 컨테이너를

재사용 하는 것이 스케쥴링의 핵심

Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019

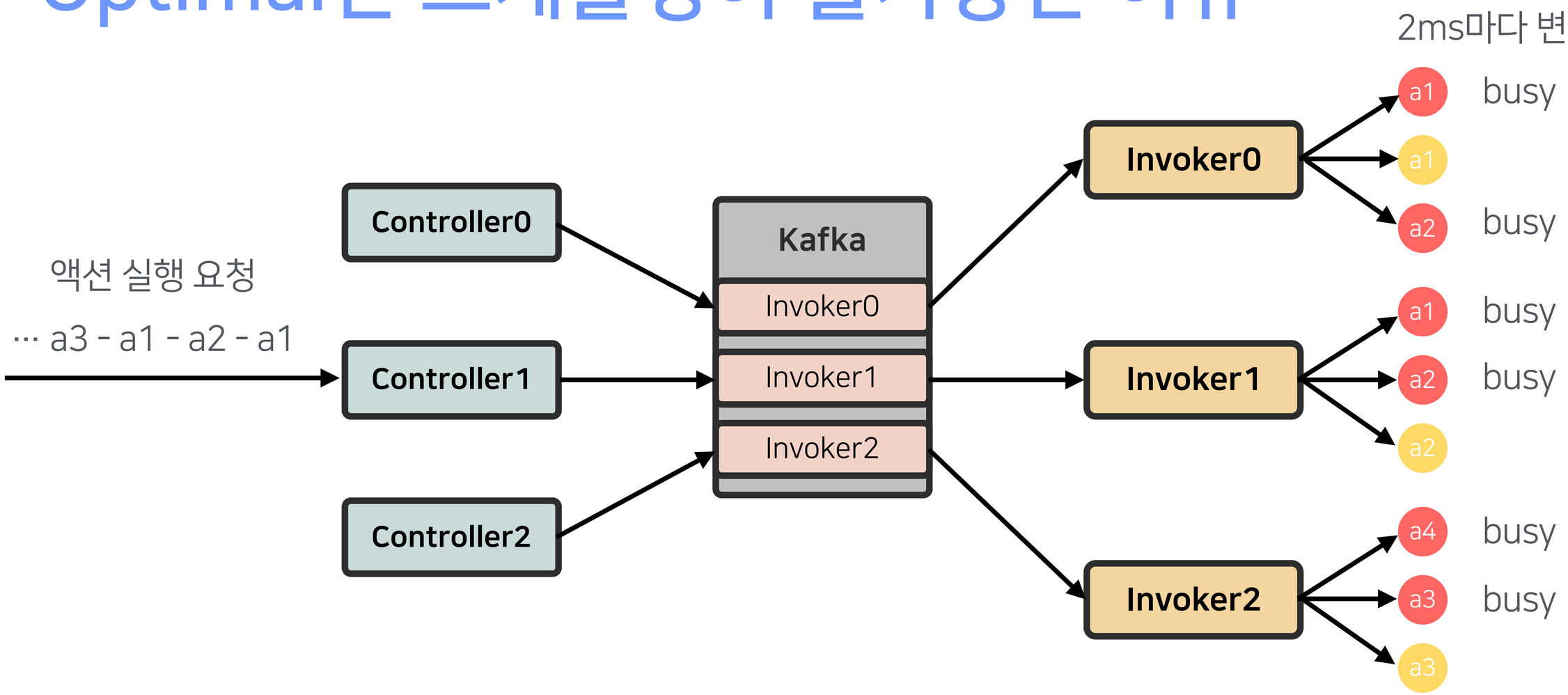


Optimal한 스케쥴링이 불가능한 이유

DEVIEW
2019

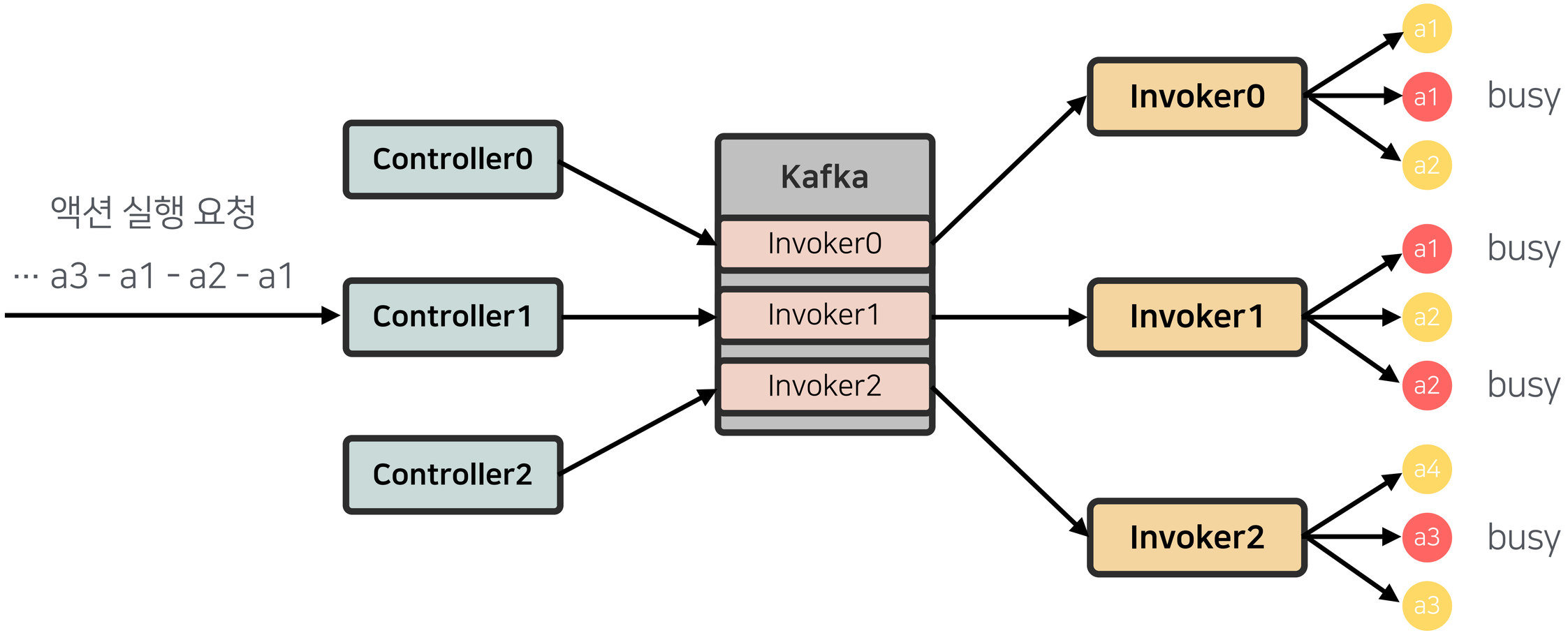
실행 시간	결과
2019. 10. 24. 18:18:06 (2ms)	success

Optimal한 스케쥴링이 불가능한 이유



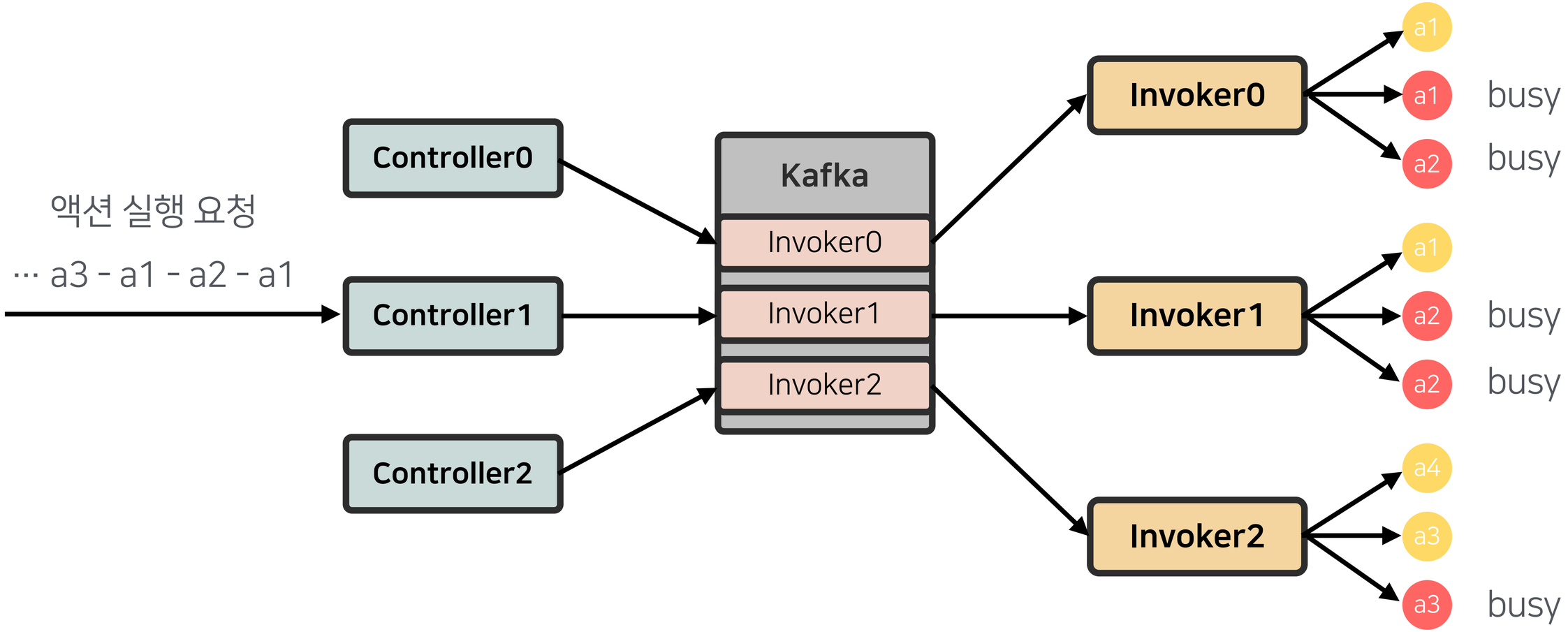
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



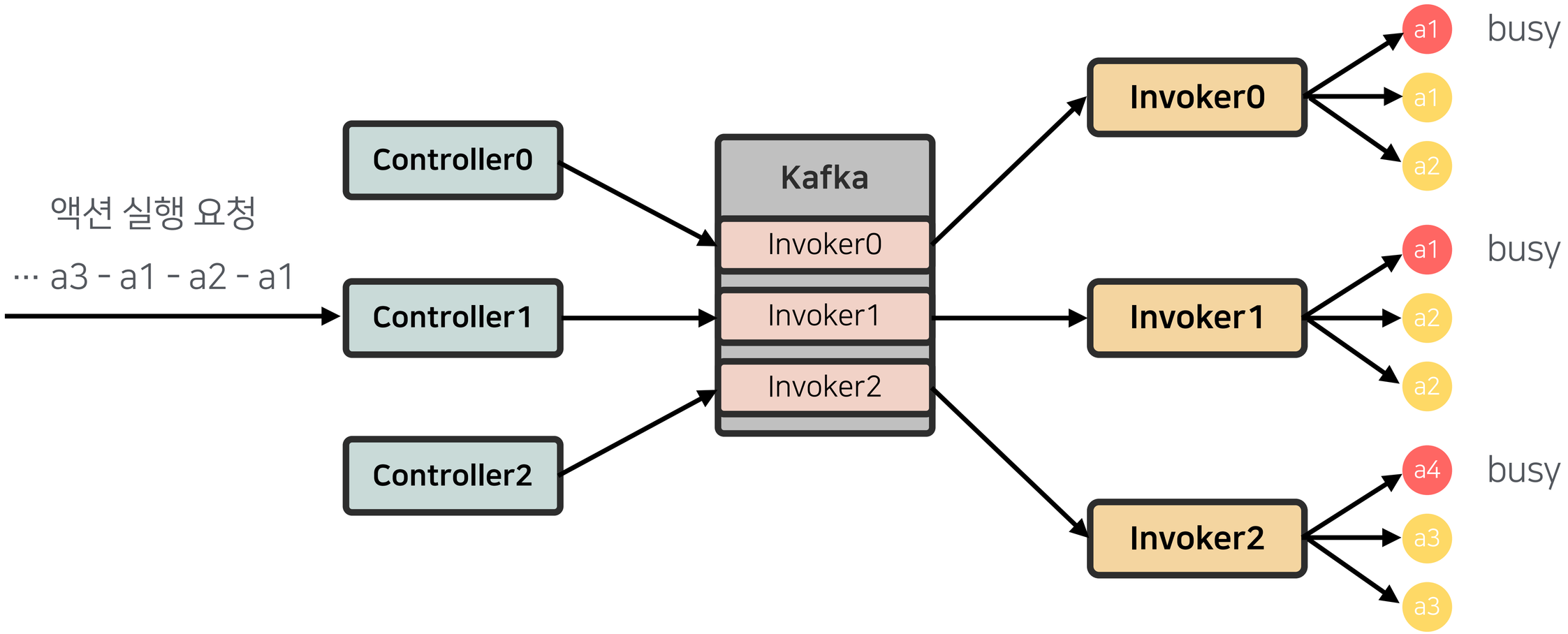
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



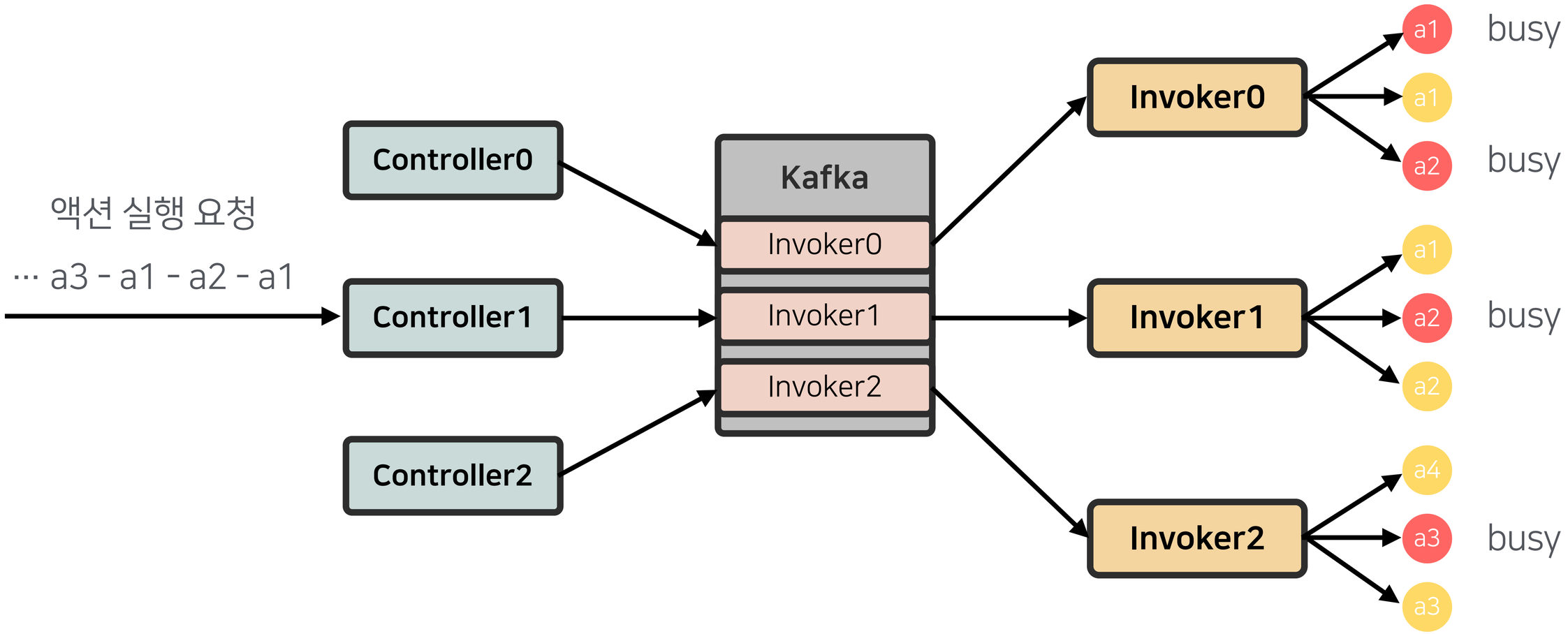
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



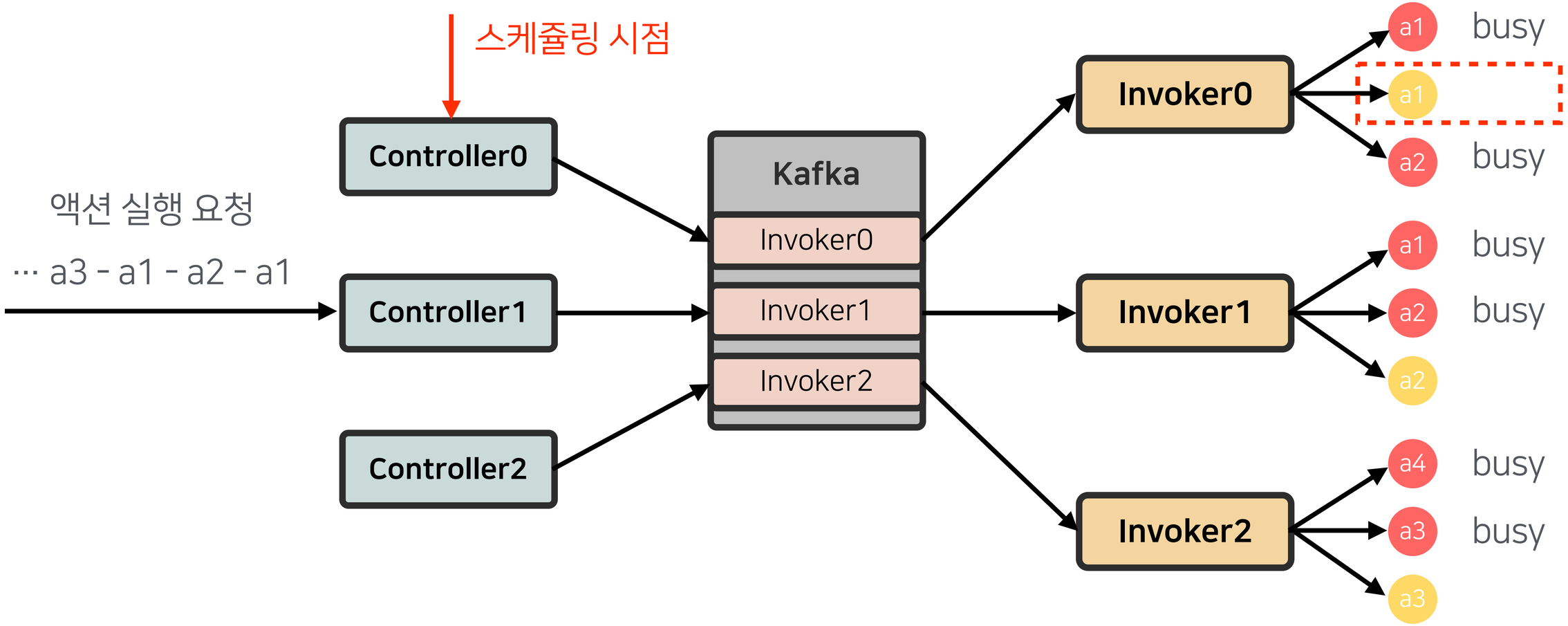
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



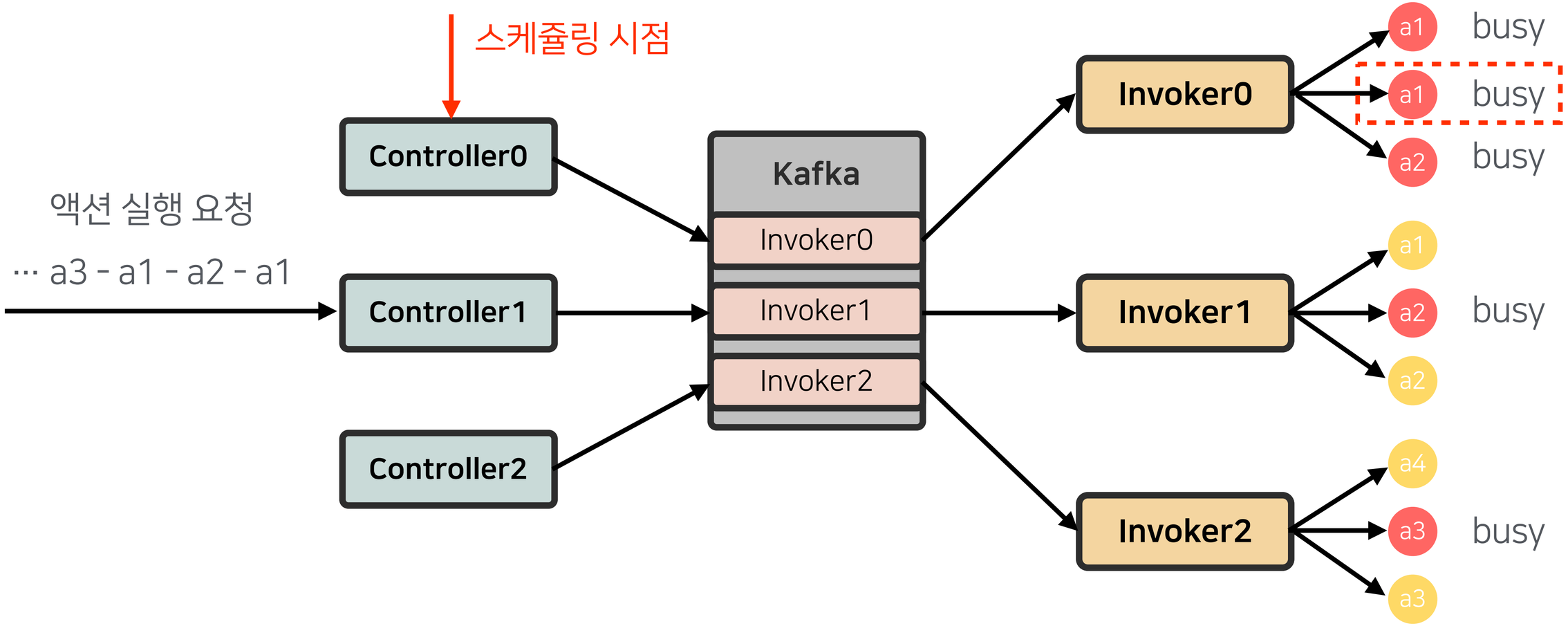
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



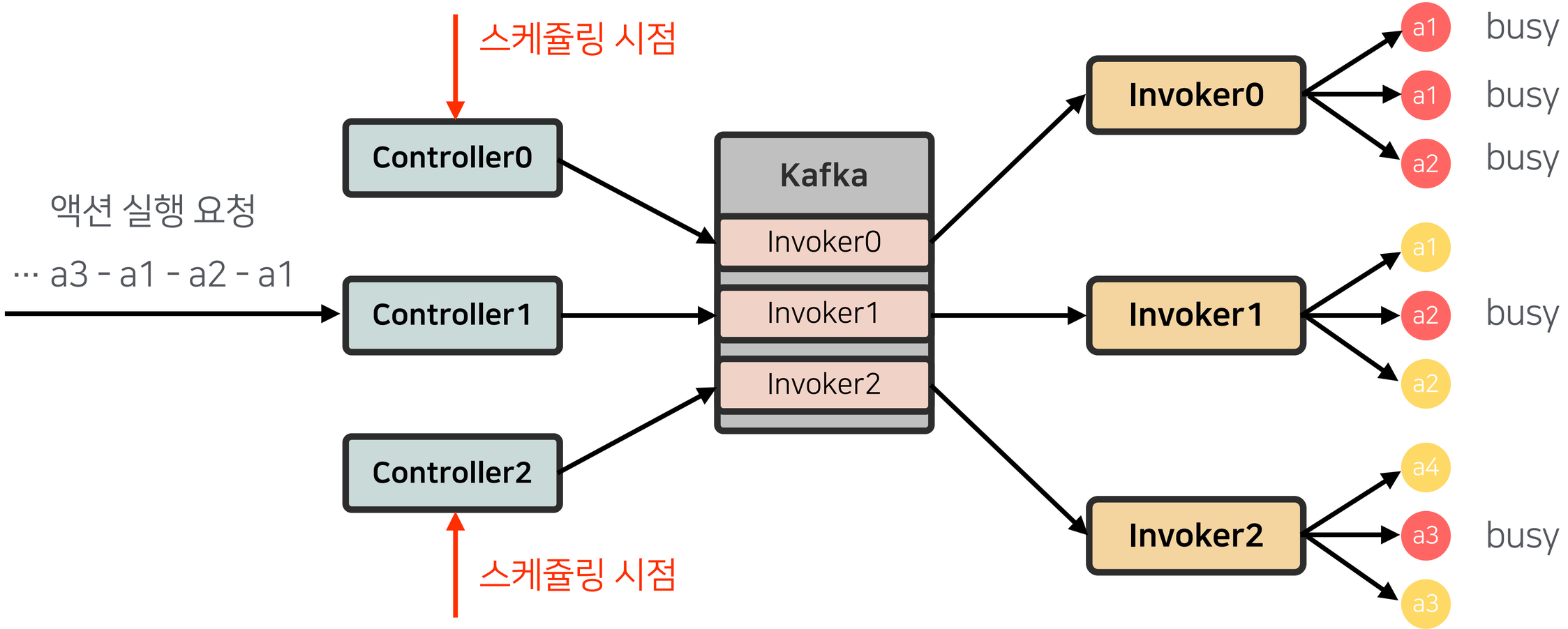
Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



Optimal한 스케쥴링이 불가능한 이유

DEVIEW 2019



Optimal한 스케쥴링이 불가능한 이유

모든 Invoker로부터 실시간으로 리소스 정보를 수집

Optimal한 스케쥴링이 불가능한 이유

모든 Invoker로부터 실시간으로 리소스 정보를 수집

다른 Controller의 스케쥴링을 고려

Optimal한 스케쥴링이 불가능한 이유

모든 Invoker로부터 실시간으로 리소스 정보를 수집

다른 Controller의 스케쥴링을 고려

최적의 위치로 실행 요청을 전송

Optimal한 스케쥴링이 불가능한 이유

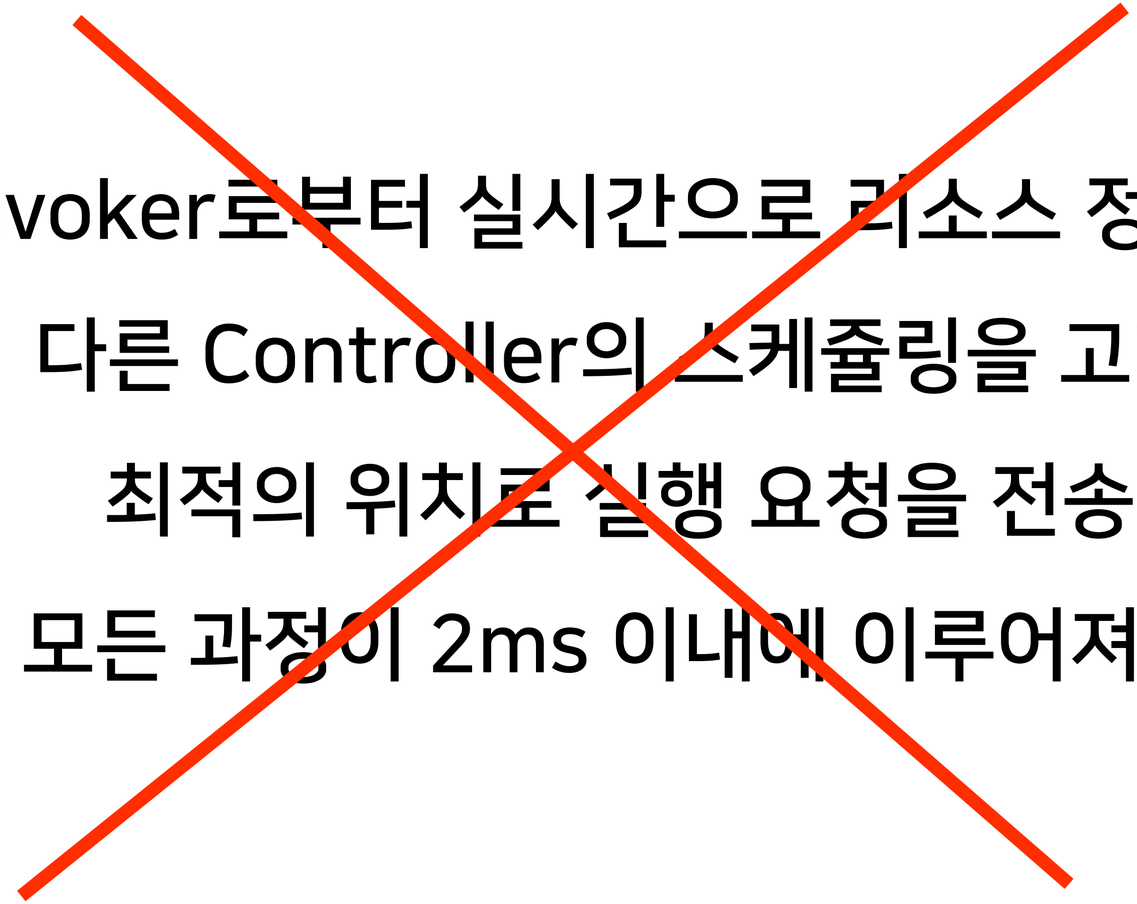
모든 Invoker로부터 실시간으로 리소스 정보를 수집

다른 Controller의 스케쥴링을 고려

최적의 위치로 실행 요청을 전송

이 모든 과정이 2ms 이내에 이루어져야 함

Optimal한 스케쥴링이 불가능한 이유



모든 Invoker로부터 실시간으로 리소스 정보를 수집

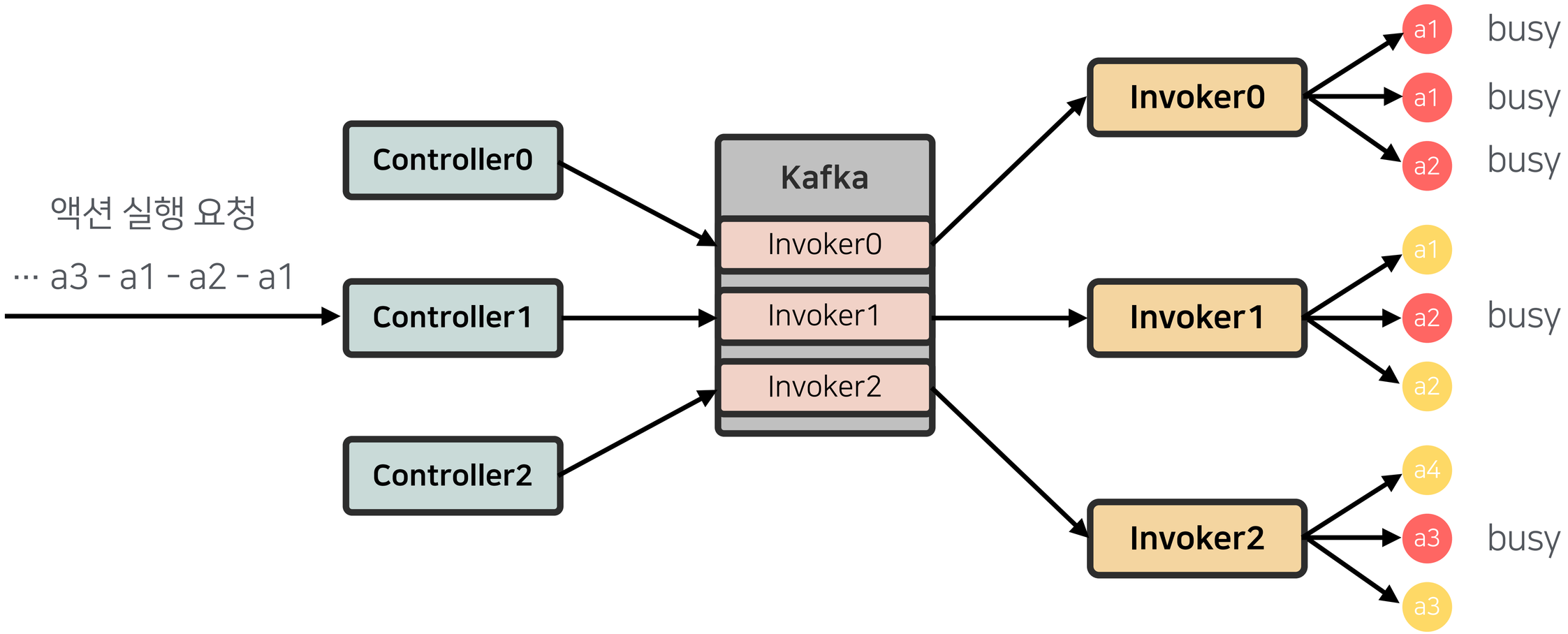
다른 Controller의 스케쥴링을 고려

최적의 위치로 실행 요청을 전송

이 모든 과정이 2ms 이내에 이루어져야 함

어떻게 할 것인가?

DEVIEW 2019



DEVIEW
2019

Apache OpenWhisk의 스케쥴링 방식

1. 액션의 위치를 미리 정해둠

Hash 함수를 통해 액션의 위치를 결정

1. Hash 함수를 통한 위치 결정

Hash(a1) = HomeInvoker

1. Hash 함수를 통한 위치 결정

DEVIEW
2019

Hash(a1) = 0

Hash(a2) = 1

Hash(a3) = 2

1. Hash 함수를 통한 위치 결정

Hash(a1) = 0

Hash(a2) = 1

Hash(a3) = 2

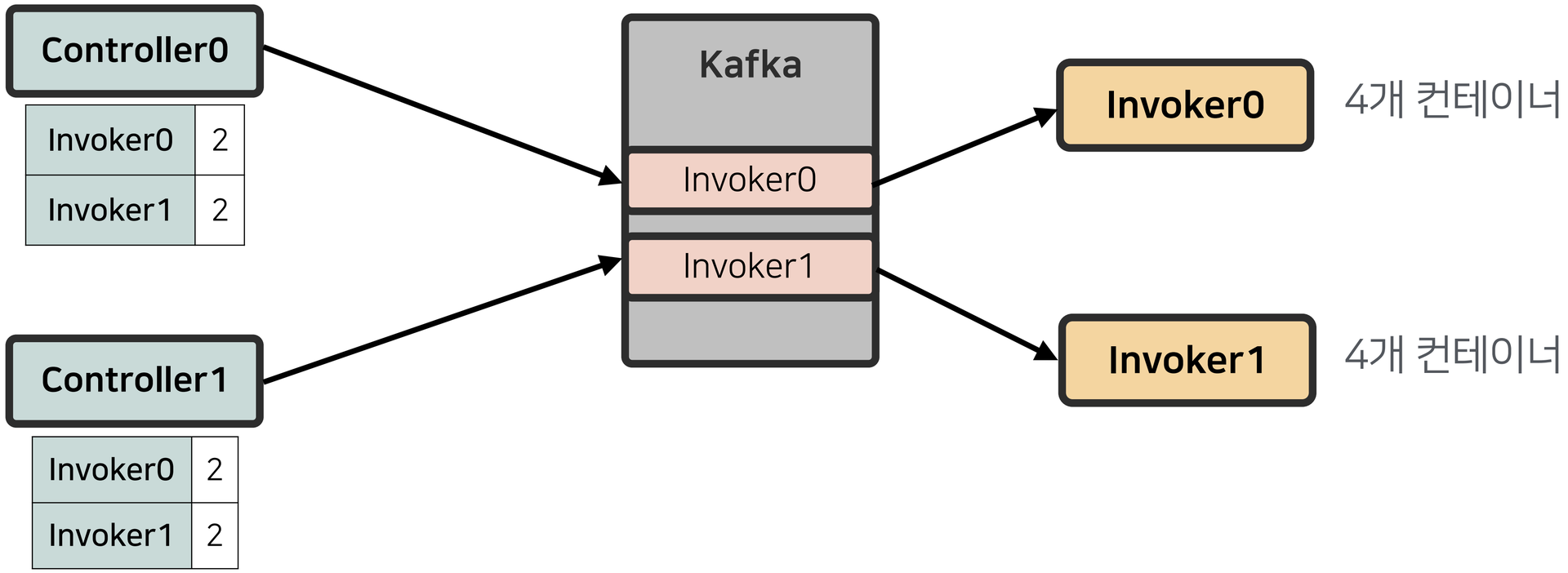
컨테이너의 위치를 고려할 필요가 없음

2. 각 컨트롤러가 리소스를 나누어 가짐

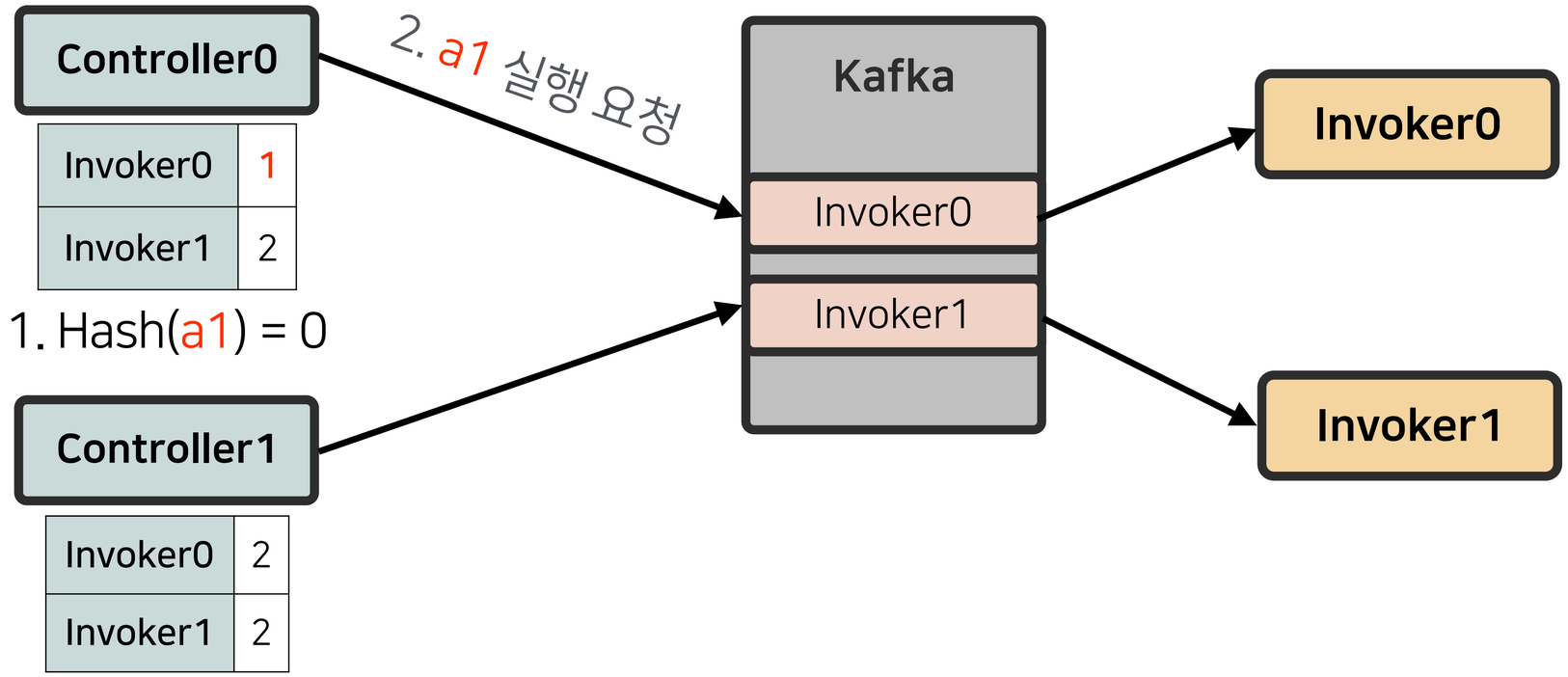
DEVIEW
2019

Invoker의 리소스 / Controller의 수

2. 각 컨트롤러가 리소스를 나누어 가짐

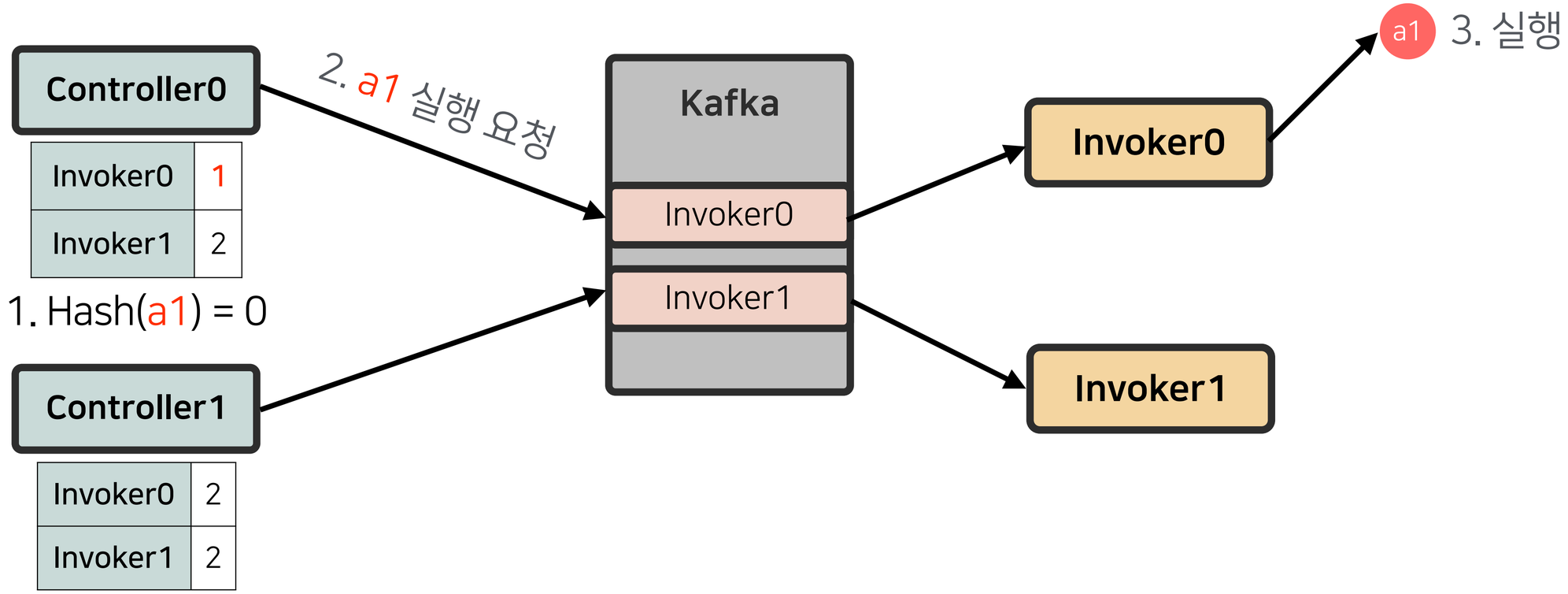


2. 각 컨트롤러가 리소스를 나누어 가짐

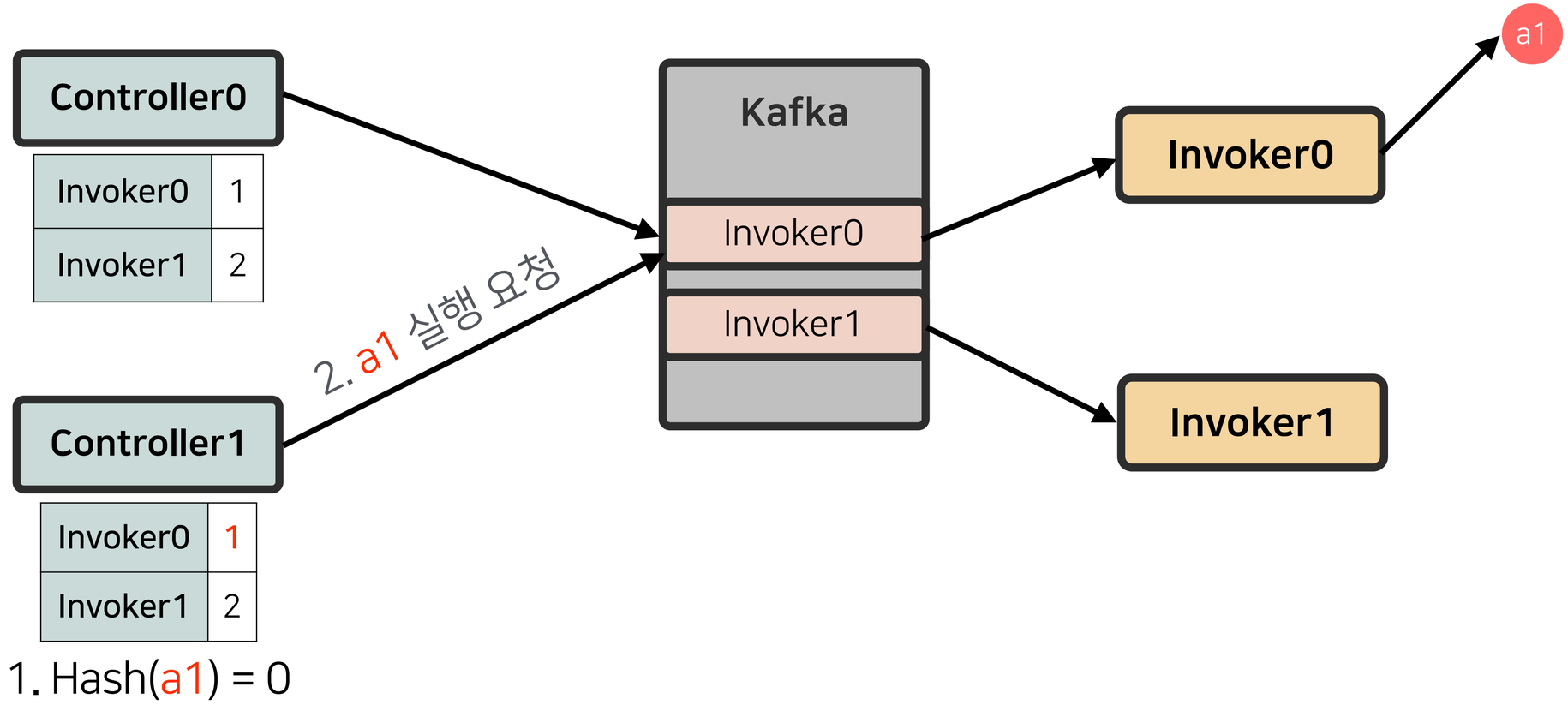


2. 각 컨트롤러가 리소스를 나누어 가짐

DEVIEW 2019

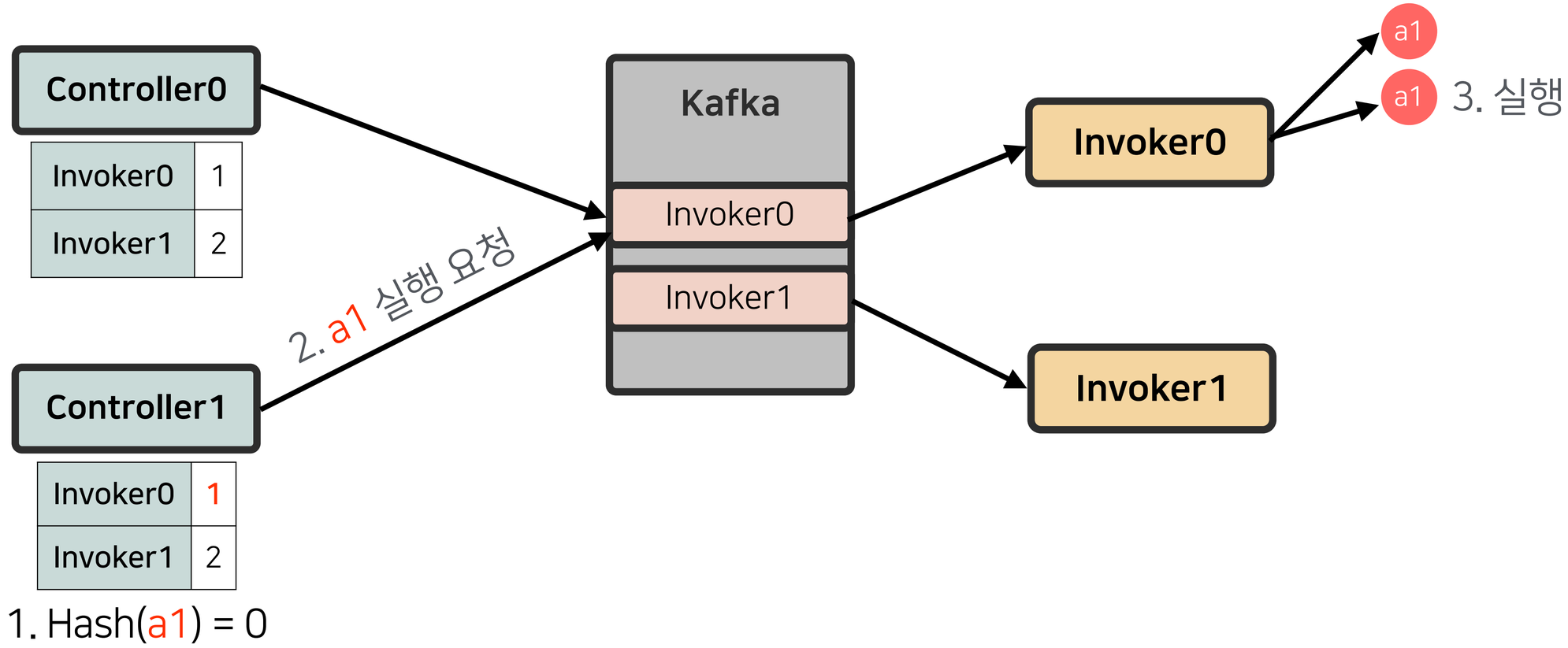


2. 각 컨트롤러가 리소스를 나누어 가짐

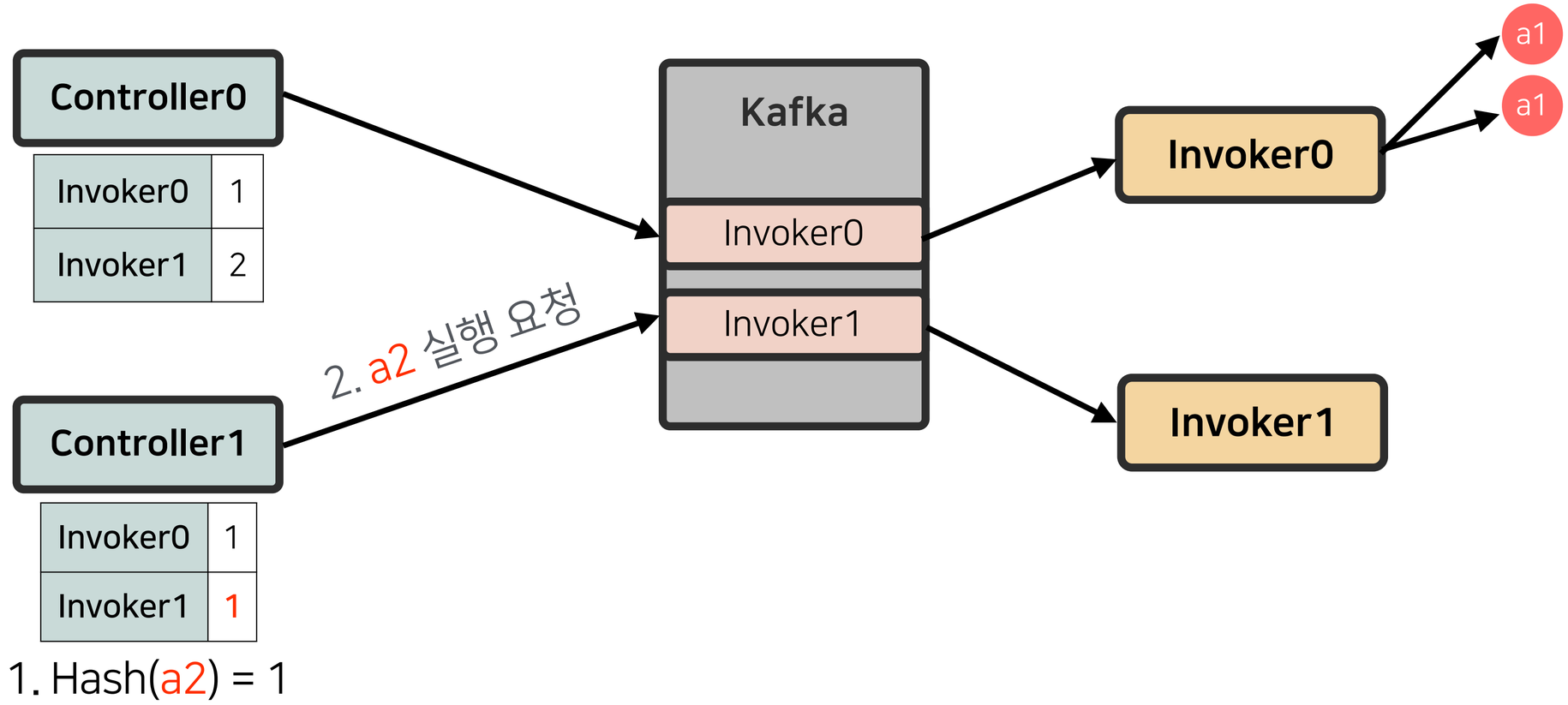


2. 각 컨트롤러가 리소스를 나누어 가짐

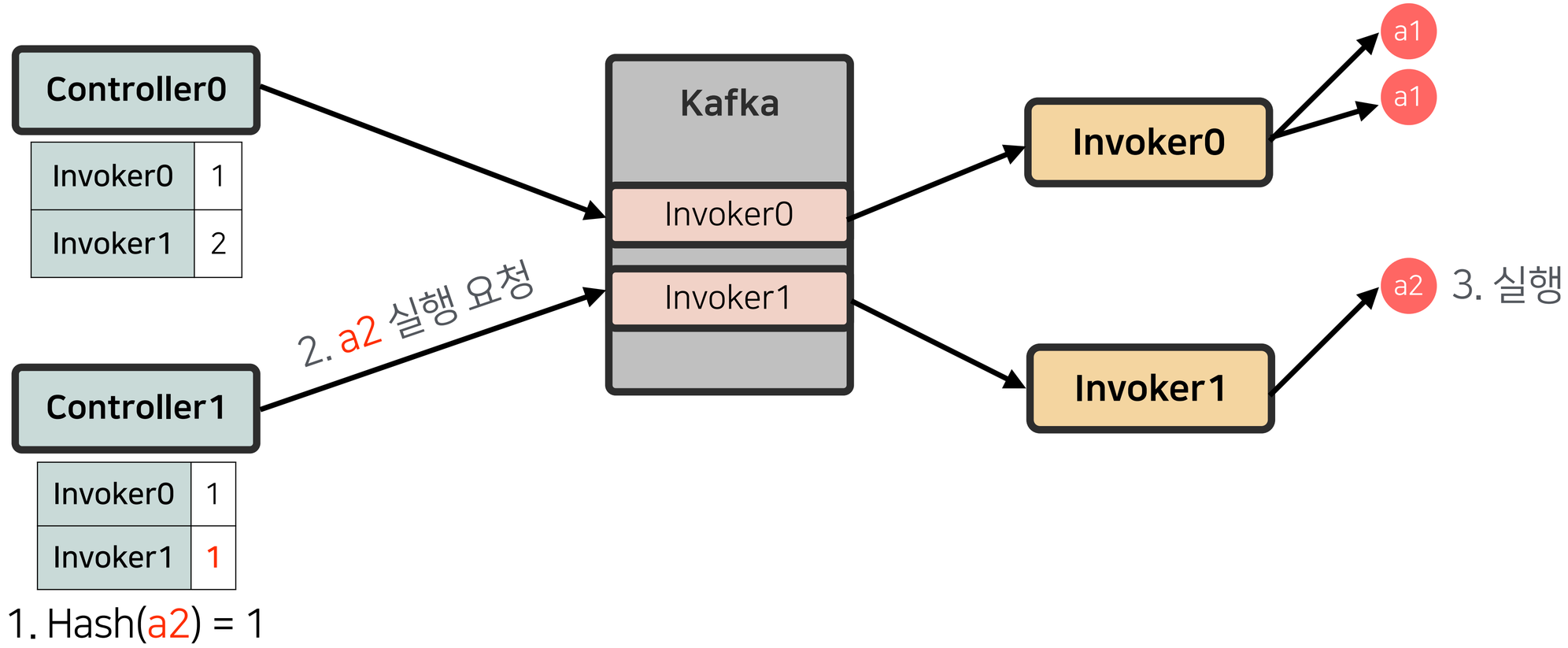
DEVIEW 2019



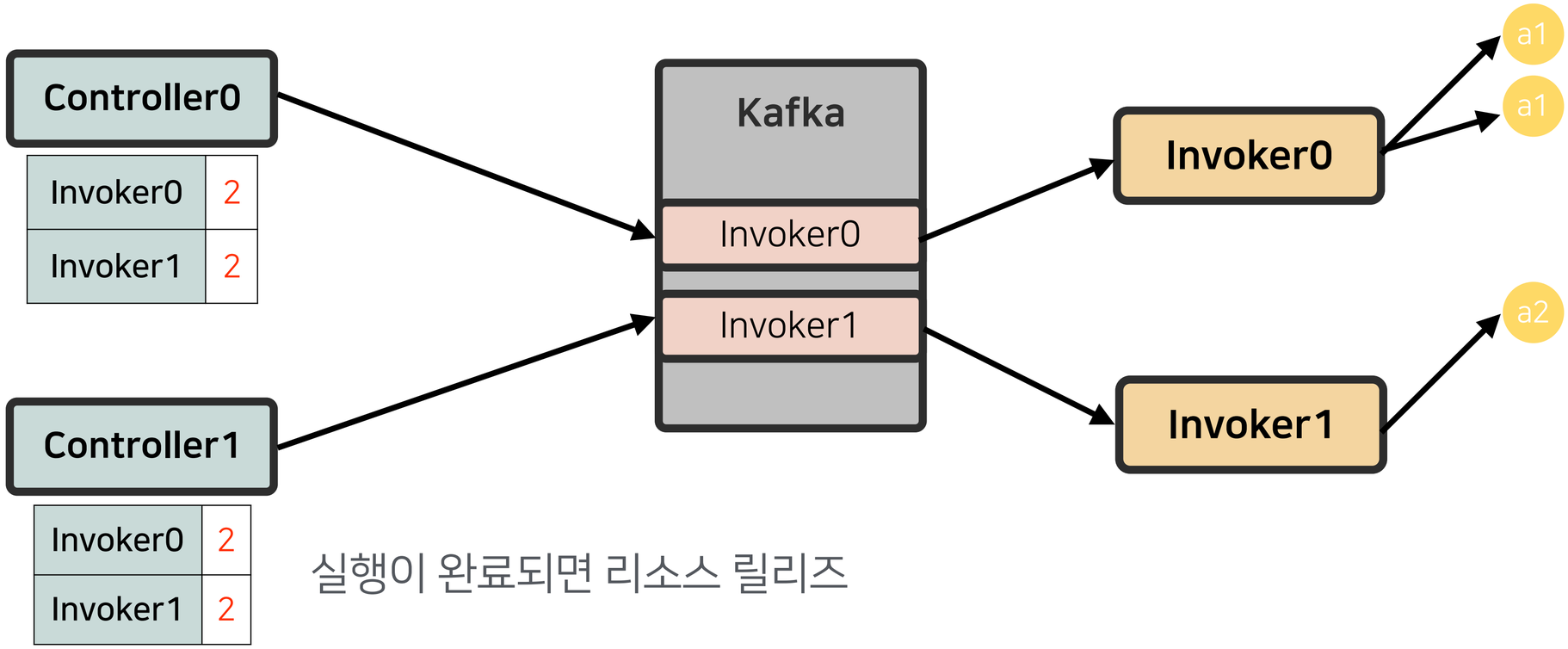
2. 각 컨트롤러가 리소스를 나누어 가짐



2. 각 컨트롤러가 리소스를 나누어 가짐



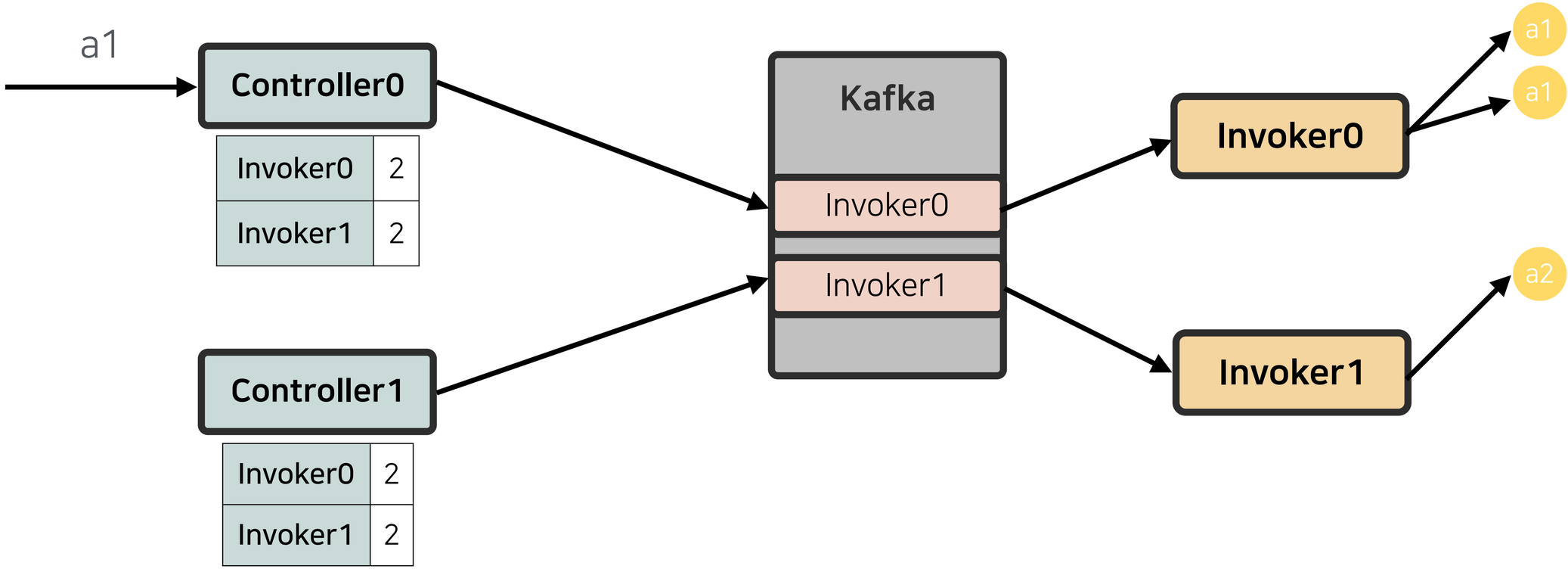
2. 각 컨트롤러가 리소스를 나누어 가짐



실행이 완료되면 리소스 릴리즈

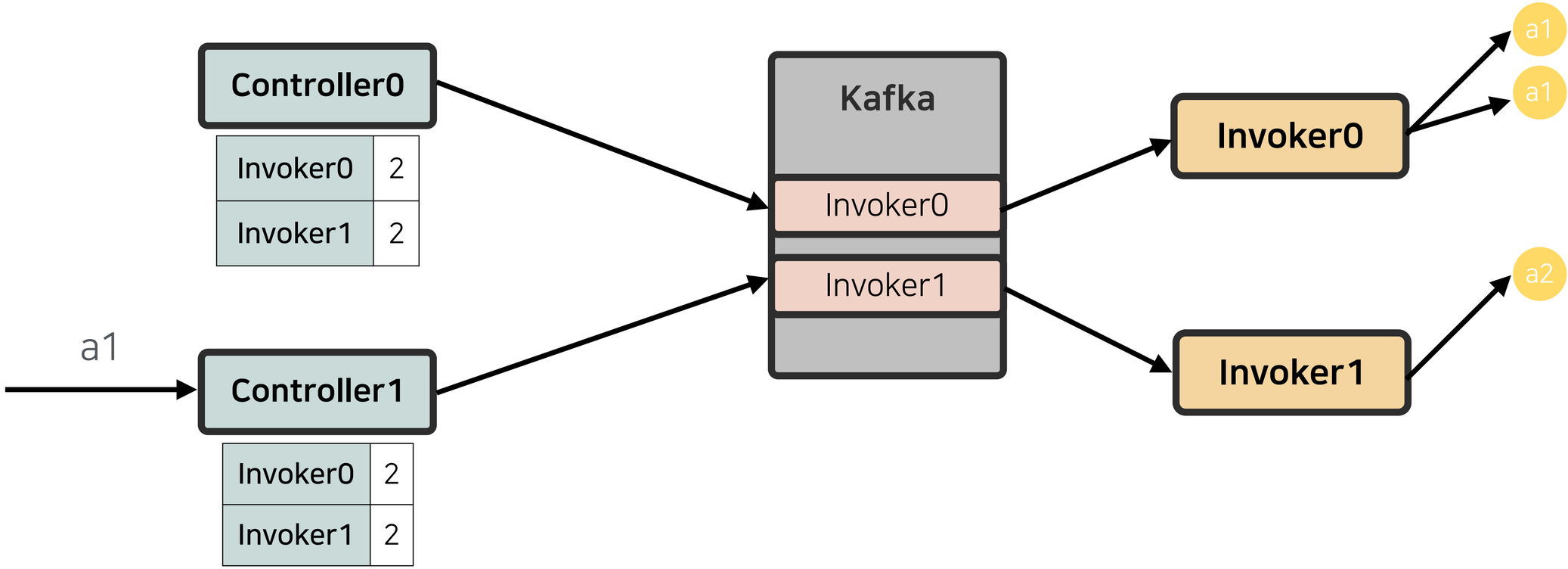
2. 각 컨트롤러가 리소스를 나누어 가짐

DEVIEW 2019

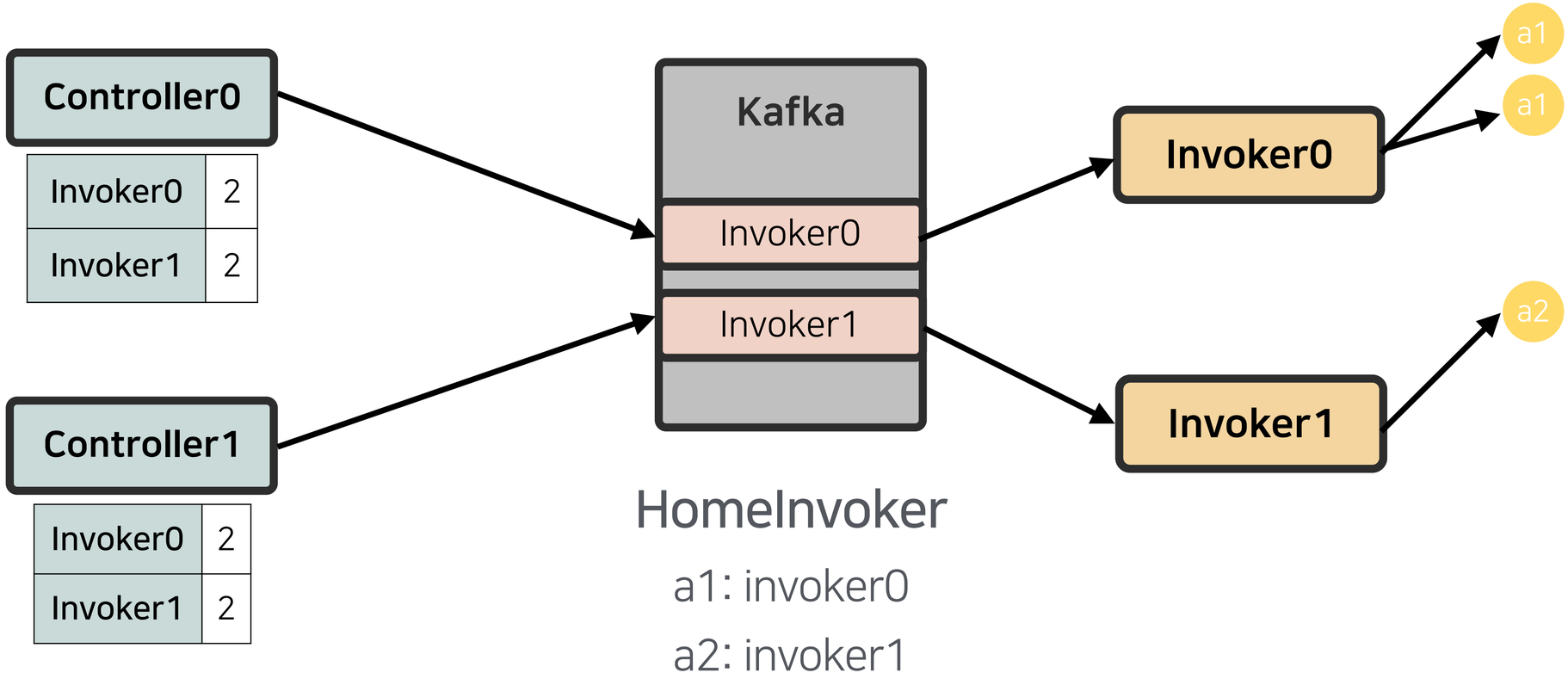


2. 각 컨트롤러가 리소스를 나누어 가짐

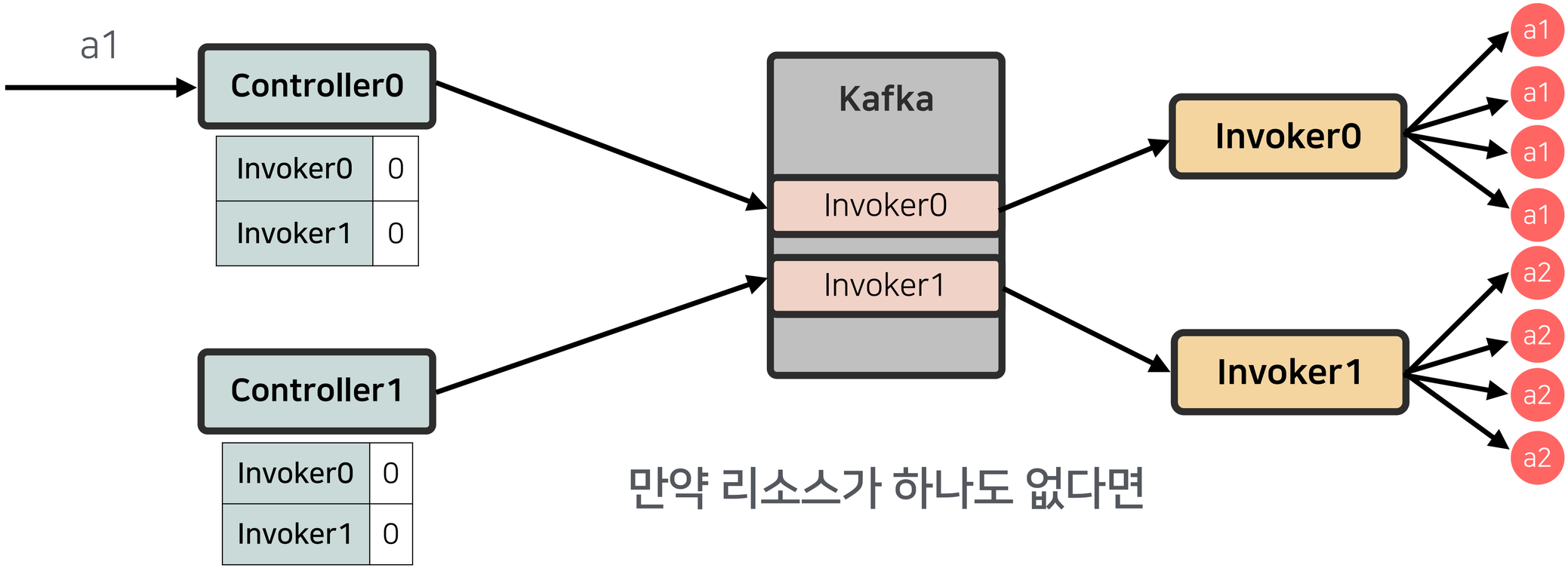
DEVIEW 2019



2. 각 컨트롤러가 리소스를 나누어 가짐

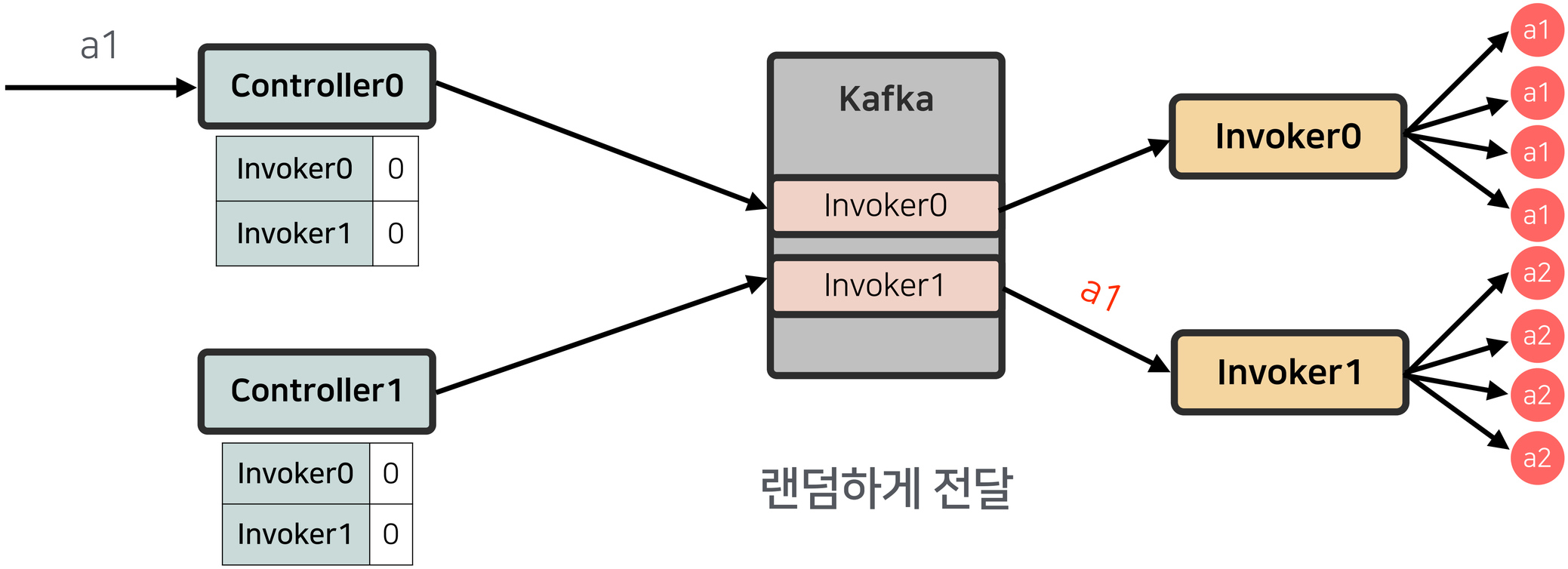


2. 각 컨트롤러가 리소스를 나누어 가짐

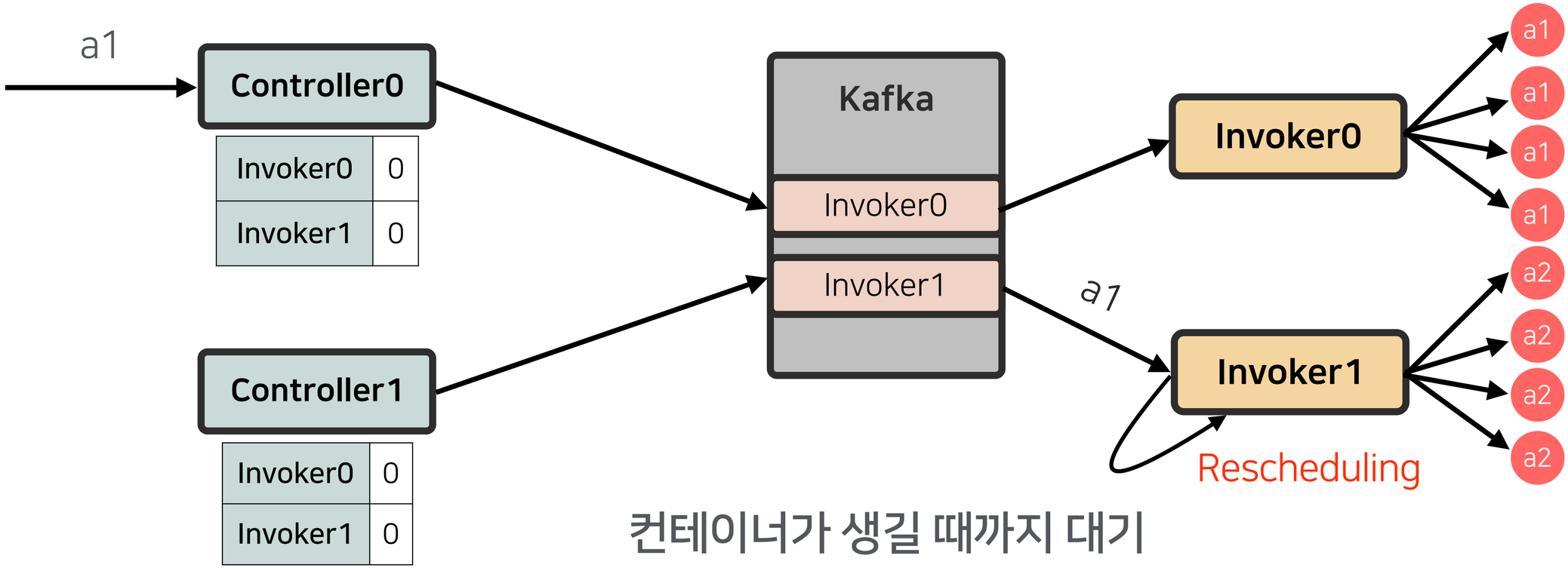


만약 리소스가 하나도 없다면

2. 각 컨트롤러가 리소스를 나누어 가짐



2. 각 컨트롤러가 리소스를 나누어 가짐



컨테이너가 생길 때까지 대기

2. 각 컨트롤러가 리소스를 나누어 가짐

다른 컨트롤러의 스케쥴링을

고려할 필요가 없음

Apache OpenWhisk의 스케쥴링

1. Hash를 통해 액션이 실행될 위치를 결정
2. 각 컨트롤러가 Invoker의 리소스를 나누어 가짐

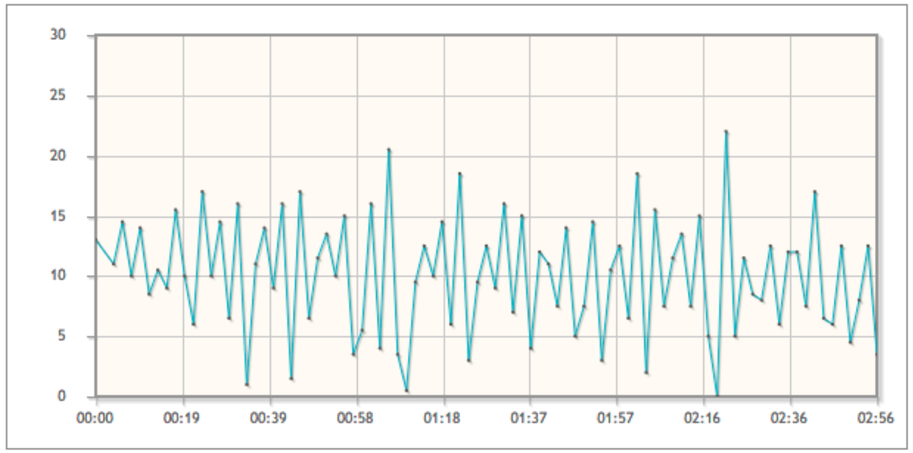
DEVIEW
2019

Docker의 성능이 OpenWhisk에 미치는 영향

Docker daemon 벤치마크 결과

DEVIEW 2019

총 Vuser	50
TPS	10.1
최고 TPS	22
평균 테스트시간	4877.28 MS
총 실행 테스트	1,776
성공한 테스트	1,776
에러	0
동작 시간	00:03:01



docker run -> rm

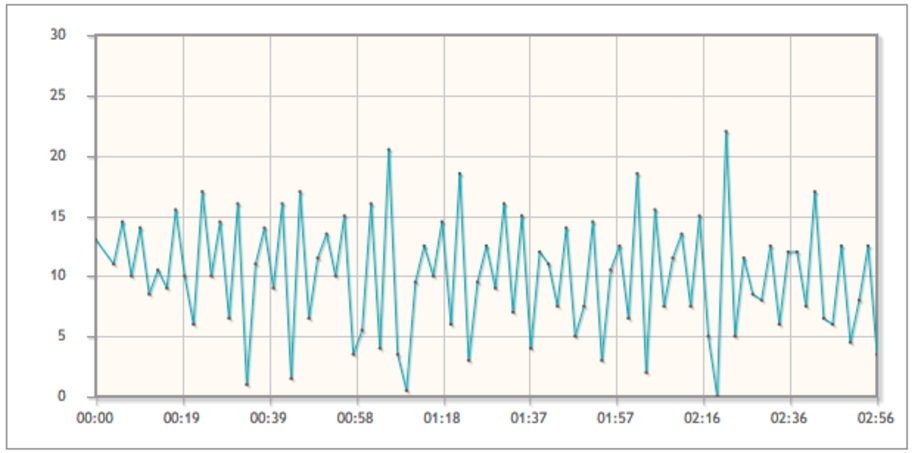
10 TPS

500ms ~ 1.3s 까지 소요

Docker daemon 벤치마크 결과

DEVIEW 2019

총 Vuser	50
TPS	10.1
최고 TPS	22
평균 테스트시간	4877.28 MS
총 실행 테스트	1,776
성공한 테스트	1,776
에러	0
동작 시간	00:03:01



docker run -> rm

10 TPS

500ms ~ 1.3s 까지 소요

요청을 Sequential하게 처리

Docker daemon 벤치마크 결과

DEVIEW 2019

3. 16 user

pause -> unpause

TPS: 37.6

```
node avg: 4249 ms
node1 avg: 4249 ms
node3 avg: 4249 ms
node2 avg: 4249 ms
node5 avg: 4249 ms
node8 avg: 4249 ms
node9 avg: 4249 ms
node4 avg: 4250 ms
node10 avg: 4250 ms
node6 avg: 4250 ms
node12 avg: 4250 ms
node11 avg: 4250 ms
node7 avg: 4250 ms
node13 avg: 4250 ms
node14 avg: 4250 ms
node15 avg: 4251 ms

total avg: 4250 ms

the number of node: 16
time per 10 action: 4250 ms
time per 1 action: 425 ms
action per 1s(TPS): (1000/425) * 16 = 37.6
```

37 TPS

50ms ~ 400ms 까지 소요

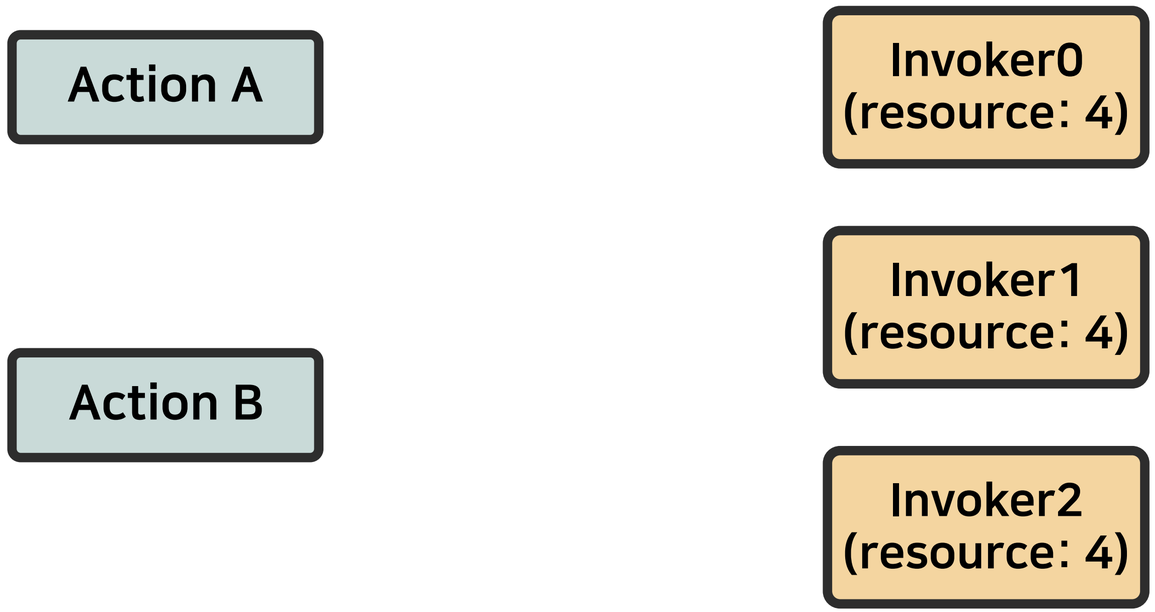
Docker daemon 벤치마크 결과

run - rm: 500ms ~ 1300ms

pause - unpause: 50ms ~ 400ms

Daemon이 요청을 sequential하게 처리

1. 액션간의 간섭


Action A
Action B
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)

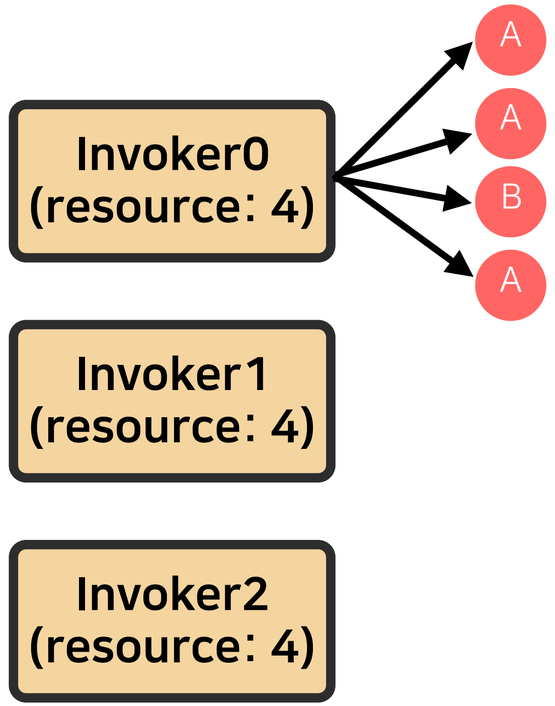
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



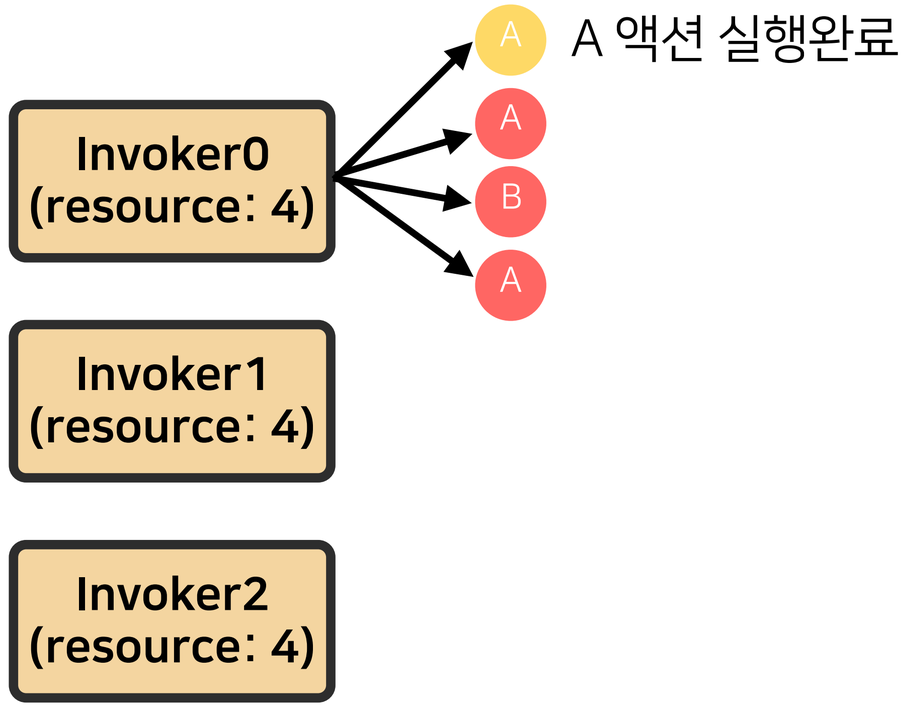
1. 액션간의 간섭

Action A

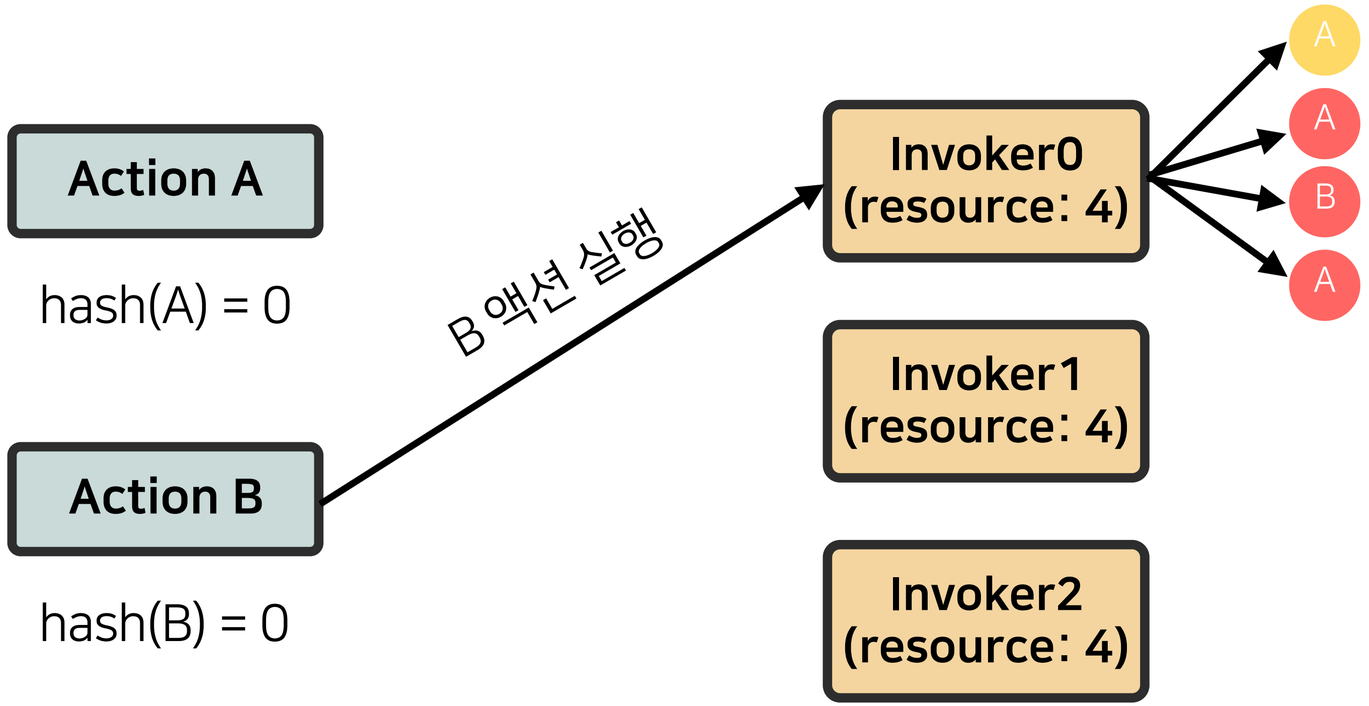
hash(A) = 0

Action B

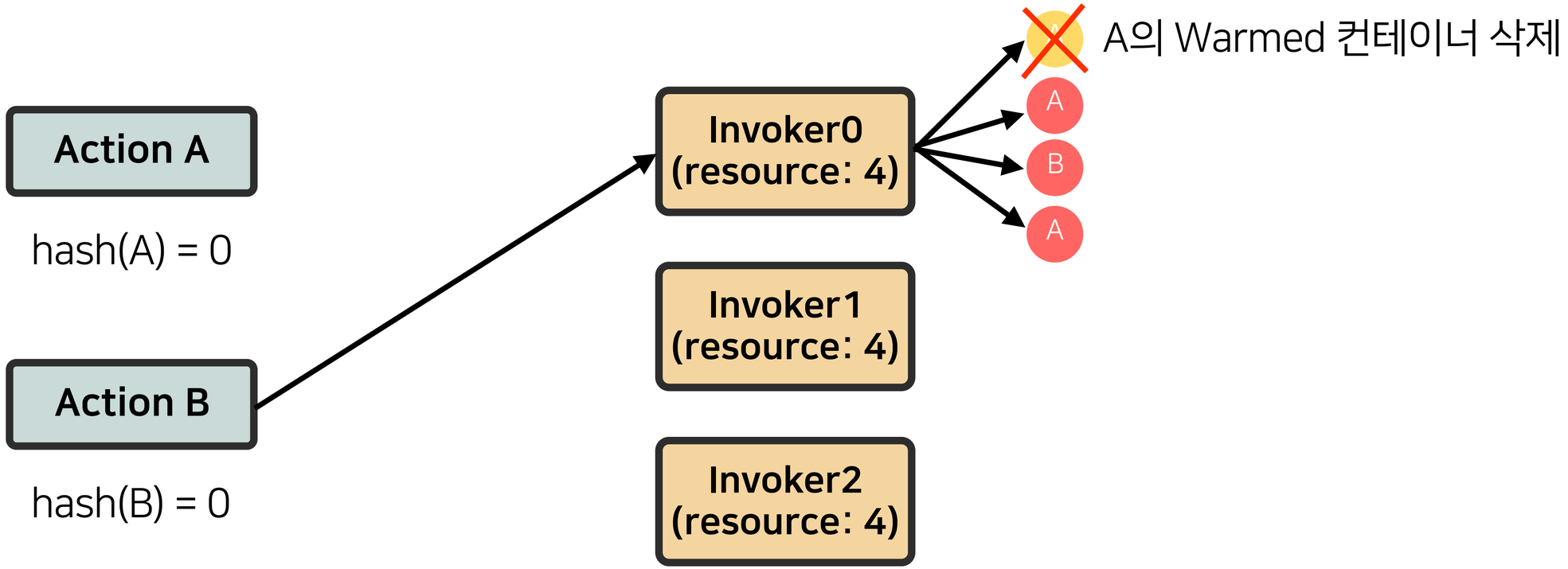
hash(B) = 0



1. 액션간의 간섭


Action A
hash(A) = 0
Action B
hash(B) = 0
B 액션 실행
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
A
A
B
A

1. 액션간의 간섭


A의 Warmed 컨테이너 삭제
Action A
hash(A) = 0
Action B
hash(B) = 0
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
A
B
A

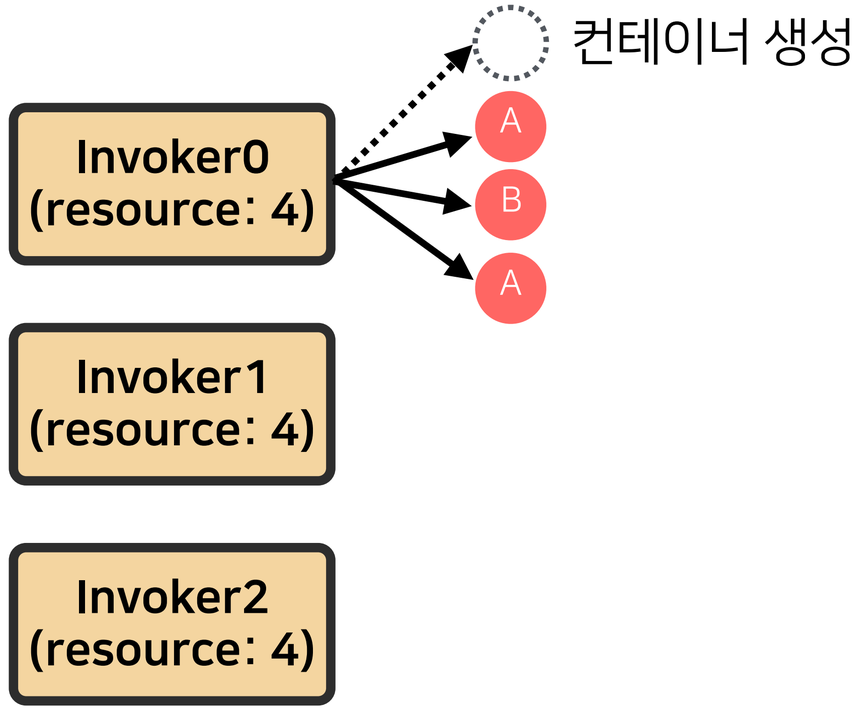
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



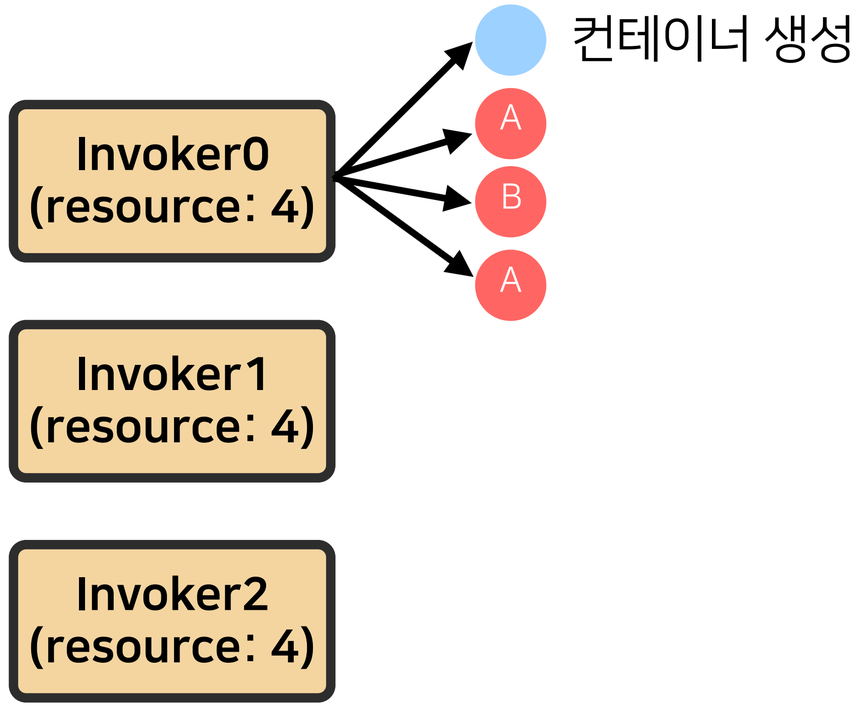
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



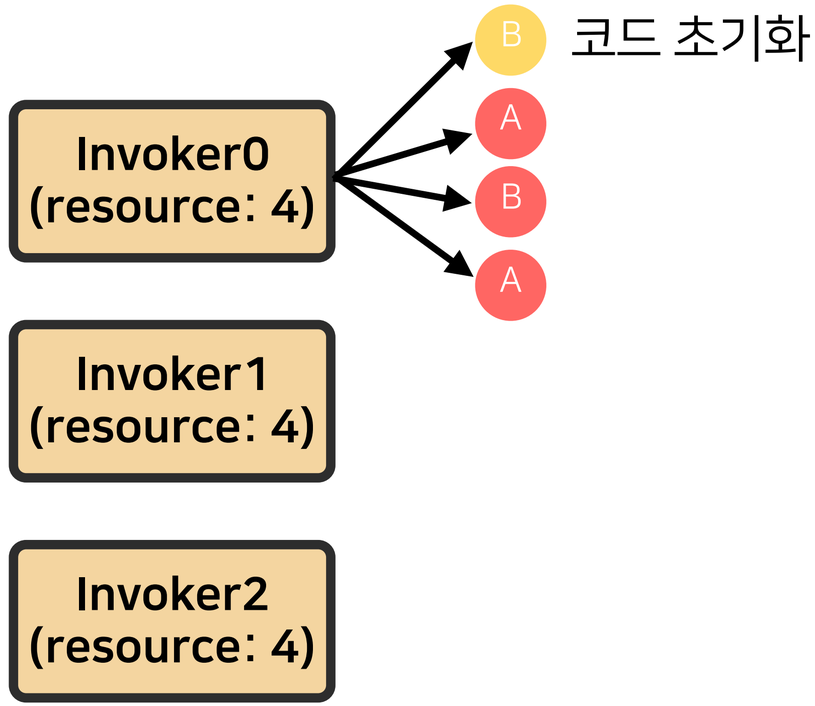
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



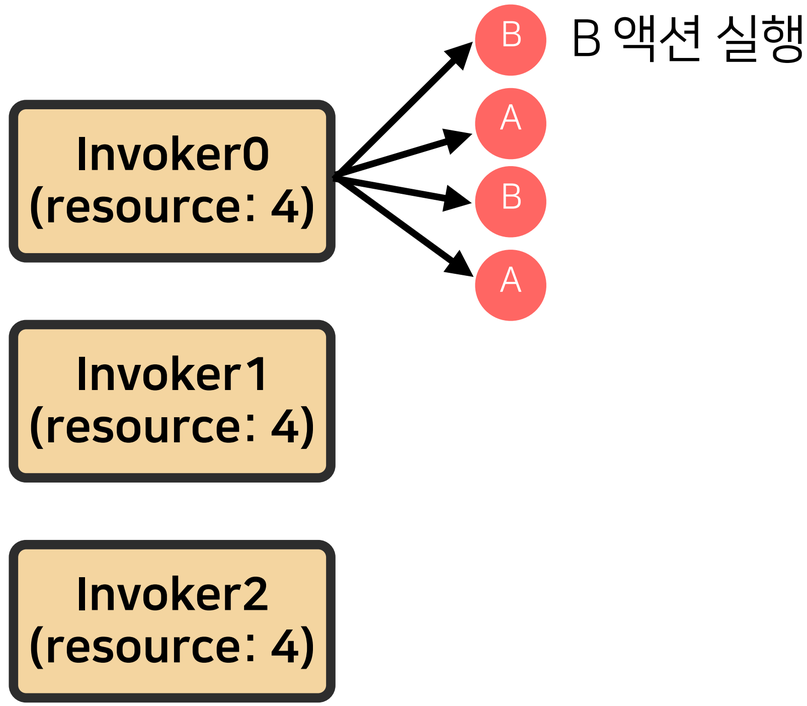
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



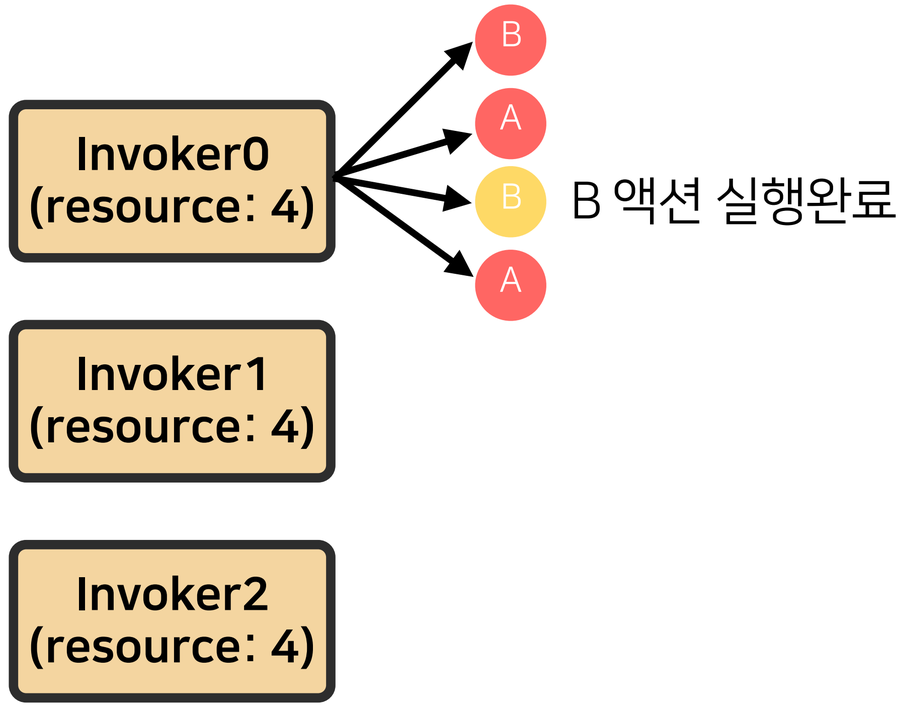
1. 액션간의 간섭

Action A

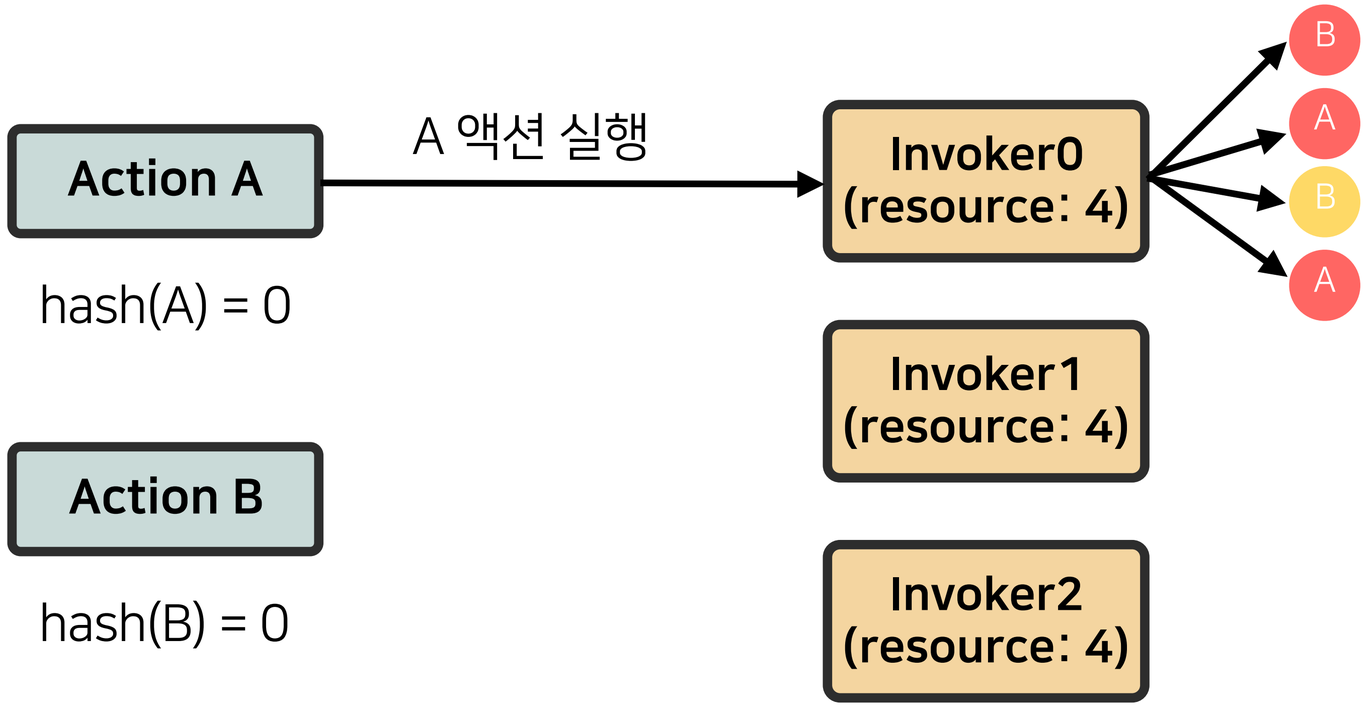
hash(A) = 0

Action B

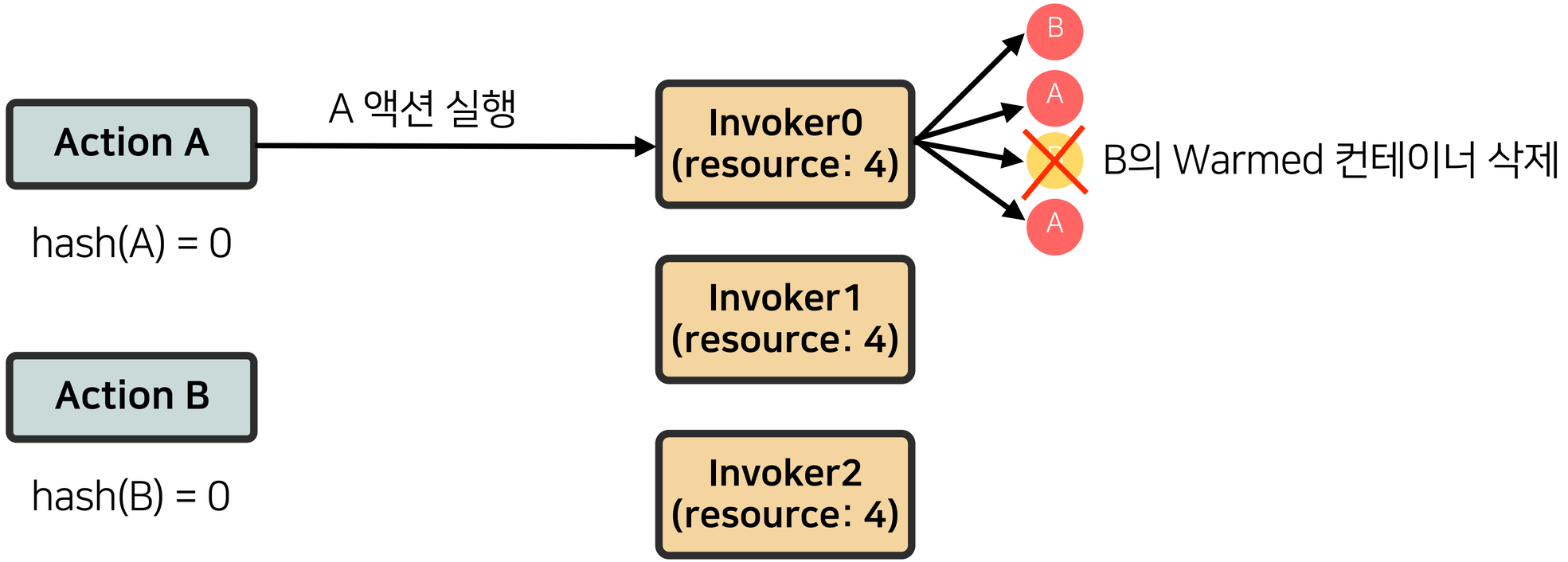
hash(B) = 0



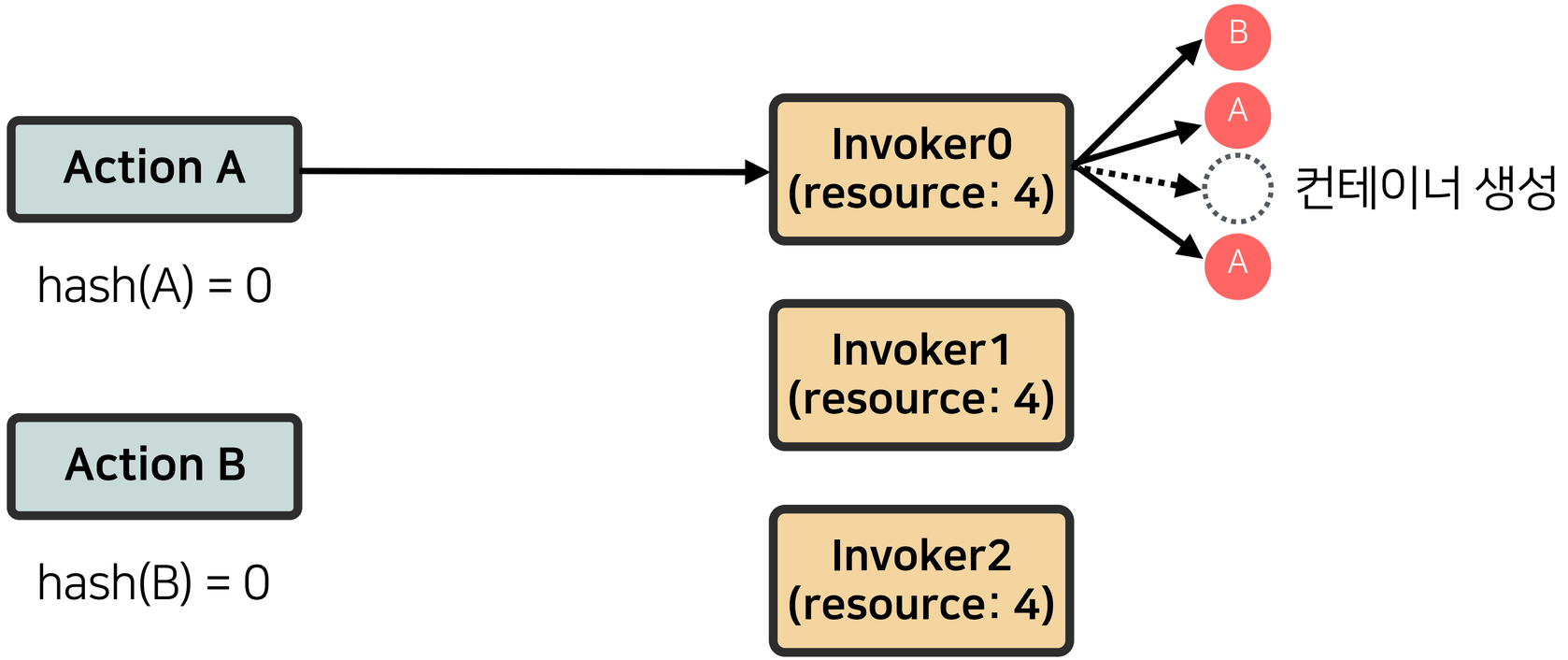
1. 액션간의 간섭


Action A
A 액션 실행
Invoker0
(resource: 4)
B
A
B
A
hash(A) = 0
Invoker1
(resource: 4)
Action B
hash(B) = 0
Invoker2
(resource: 4)

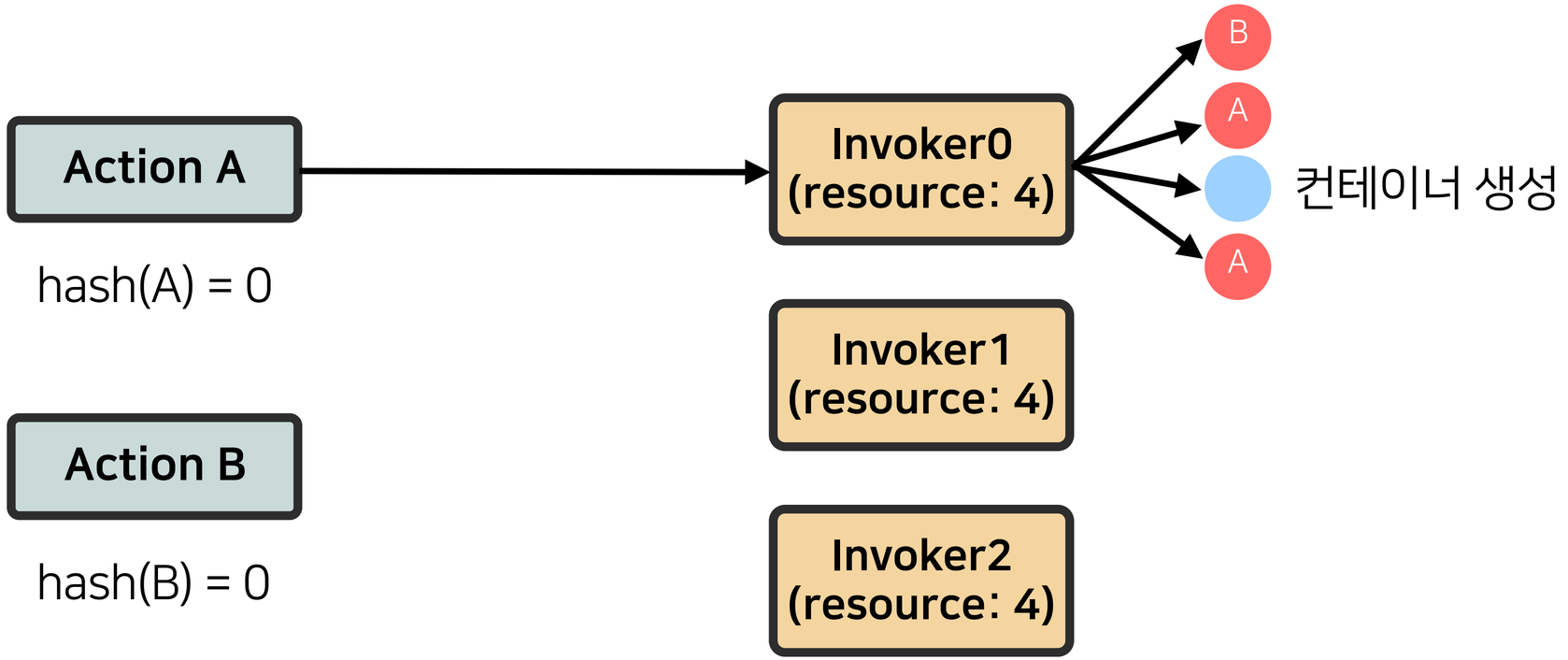
1. 액션간의 간섭


Action A
A 액션 실행
Invoker0
(resource: 4)
B
A
A
B의 Warmed 컨테이너 삭제
hash(A) = 0
Invoker1
(resource: 4)
Action B
hash(B) = 0
Invoker2
(resource: 4)

1. 액션간의 간섭

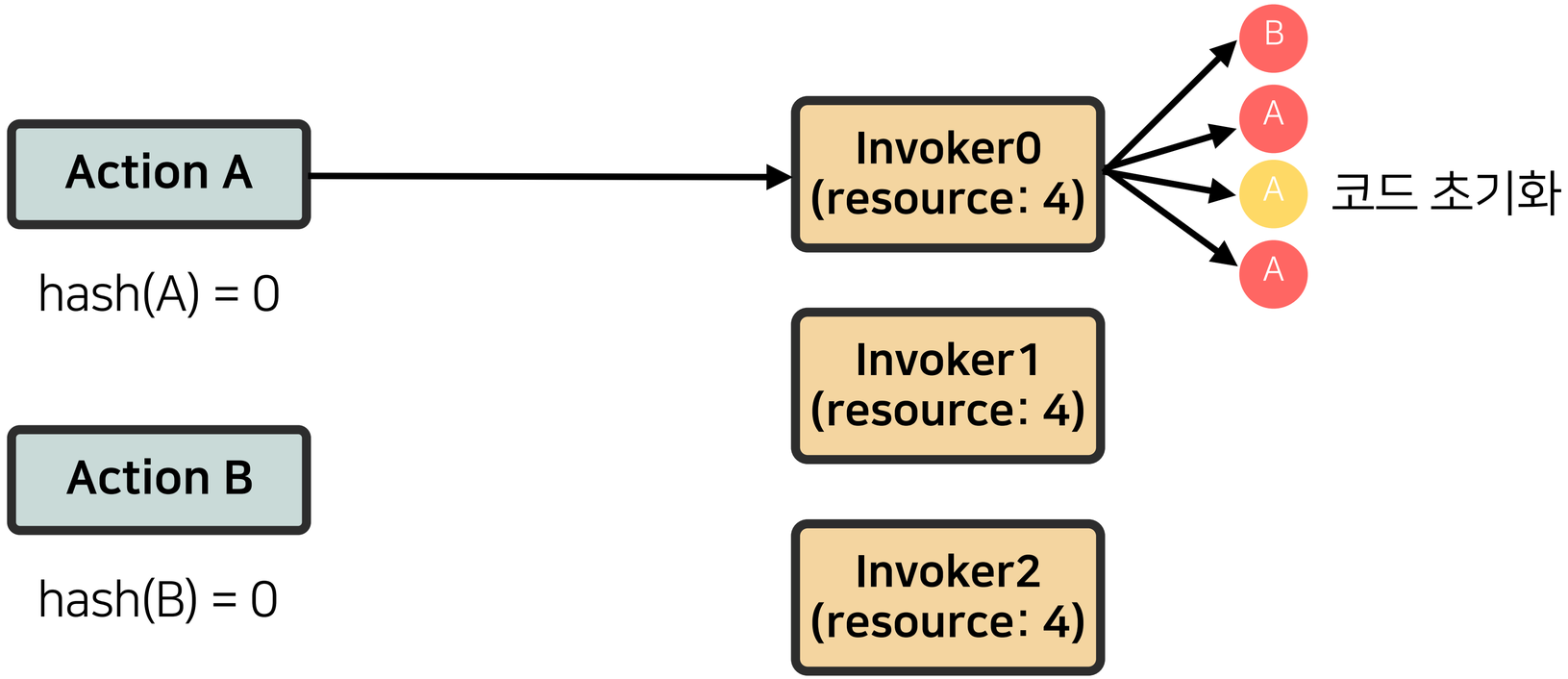

Action A
hash(A) = 0
Action B
hash(B) = 0
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
B
A
A
컨테이너 생성

1. 액션간의 간섭

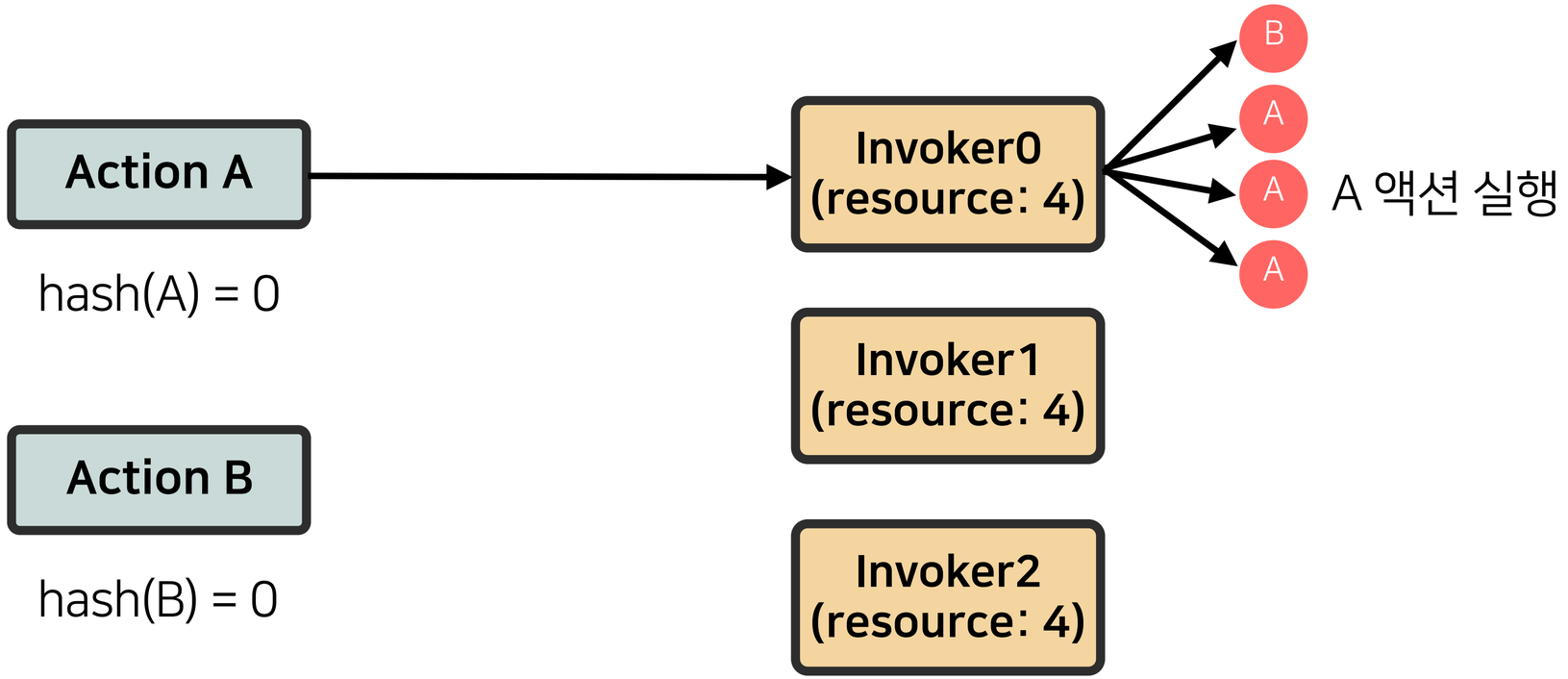

Action A
hash(A) = 0
Action B
hash(B) = 0
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
B
A
A
컨테이너 생성

1. 액션간의 간섭

DEVIEW
2019



1. 액션간의 간섭


Action A
hash(A) = 0
Action B
hash(B) = 0
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
B
A
A
A
A 액션 실행

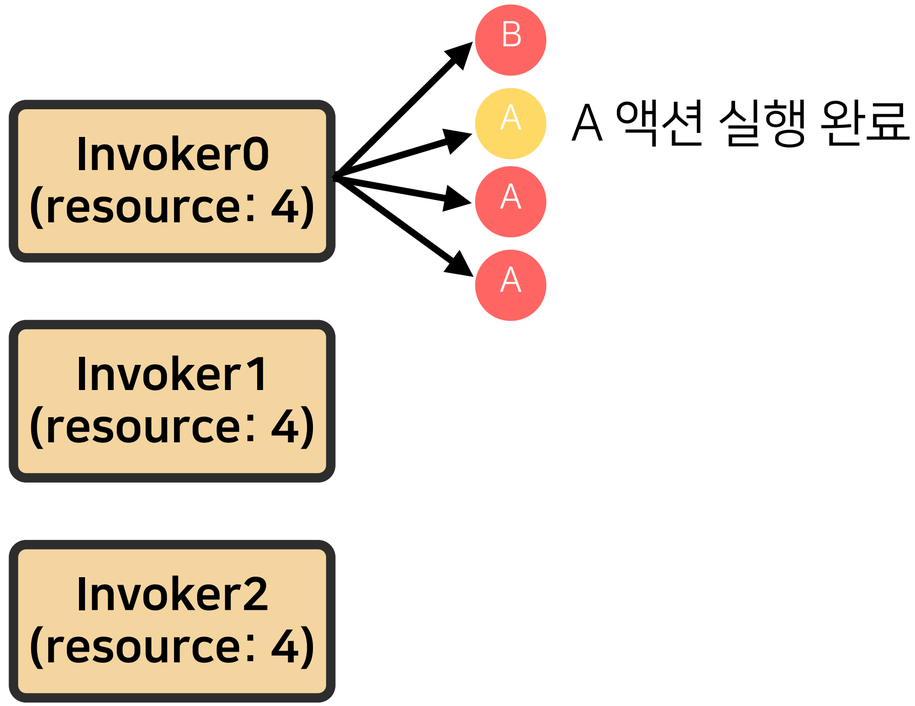
1. 액션간의 간섭

Action A

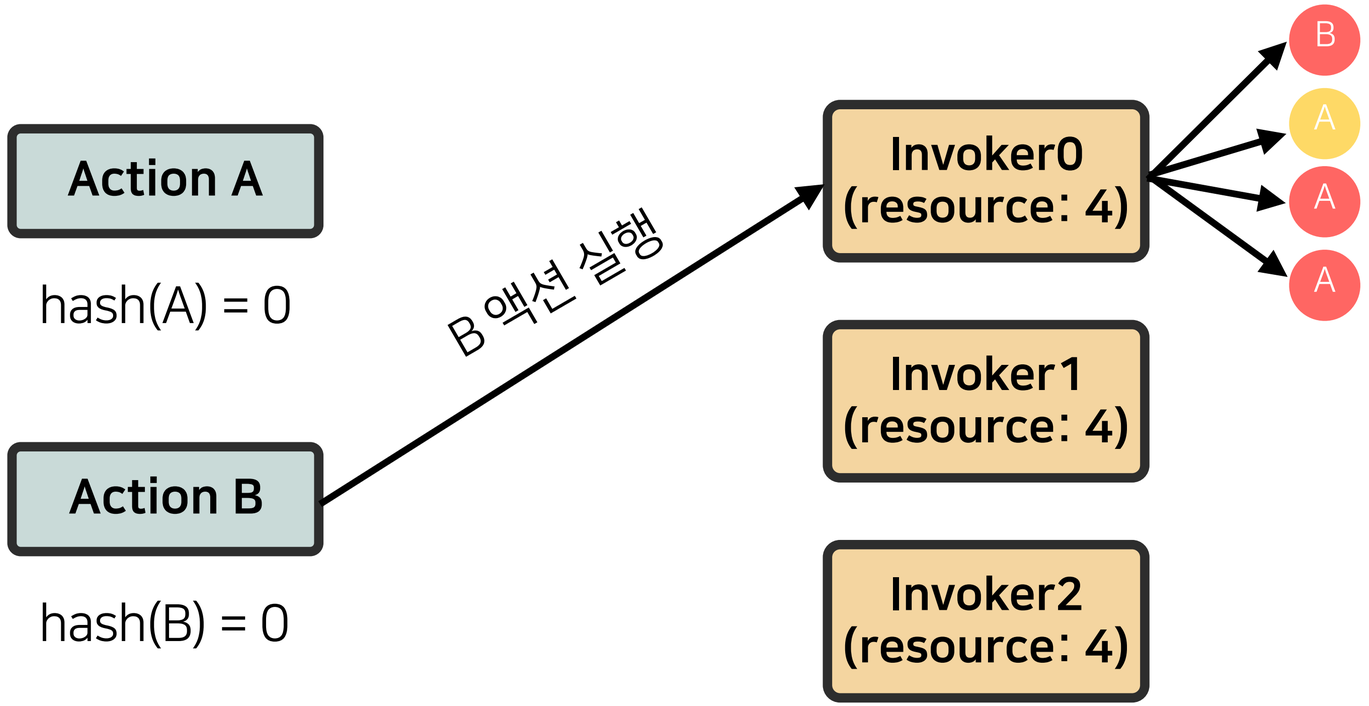
hash(A) = 0

Action B

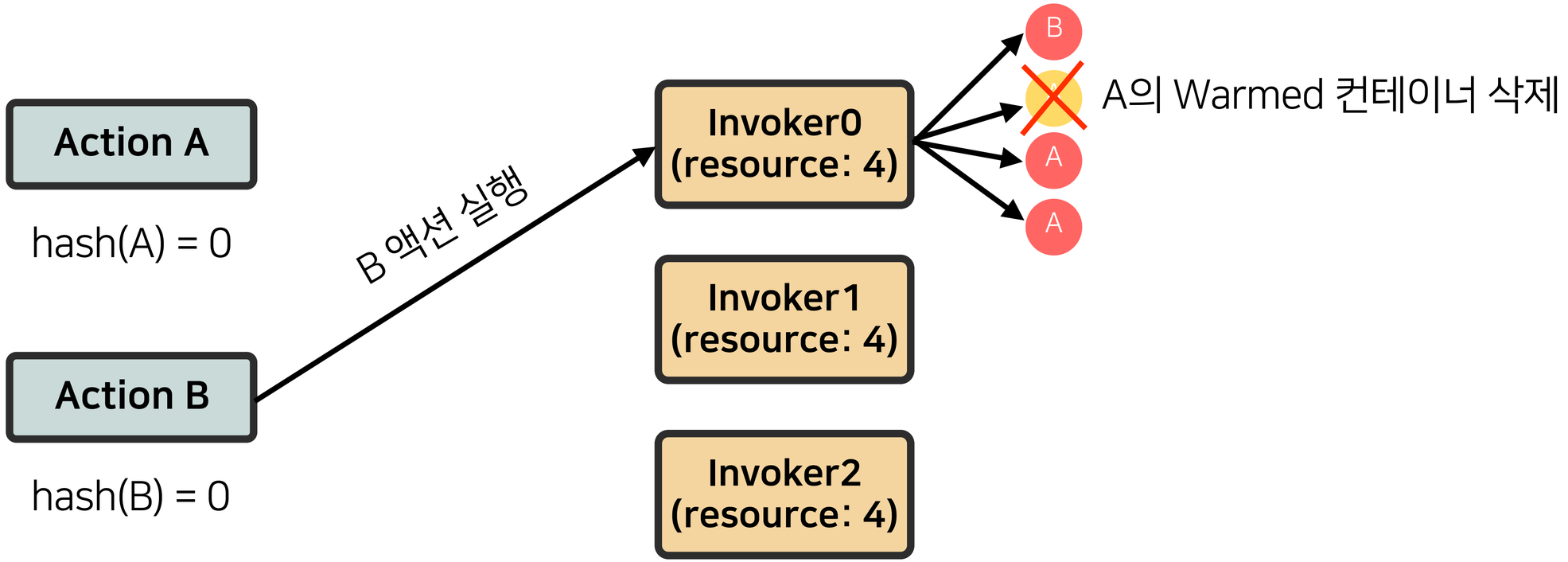
hash(B) = 0



1. 액션간의 간섭


Action A
hash(A) = 0
Action B
hash(B) = 0
B 액션 실행
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
B
A
A
A

1. 액션간의 간섭


Action A
hash(A) = 0
Action B
hash(B) = 0
B 액션 실행
Invoker0
(resource: 4)
Invoker1
(resource: 4)
Invoker2
(resource: 4)
B
A
A
A의 Warmed 컨테이너 삭제

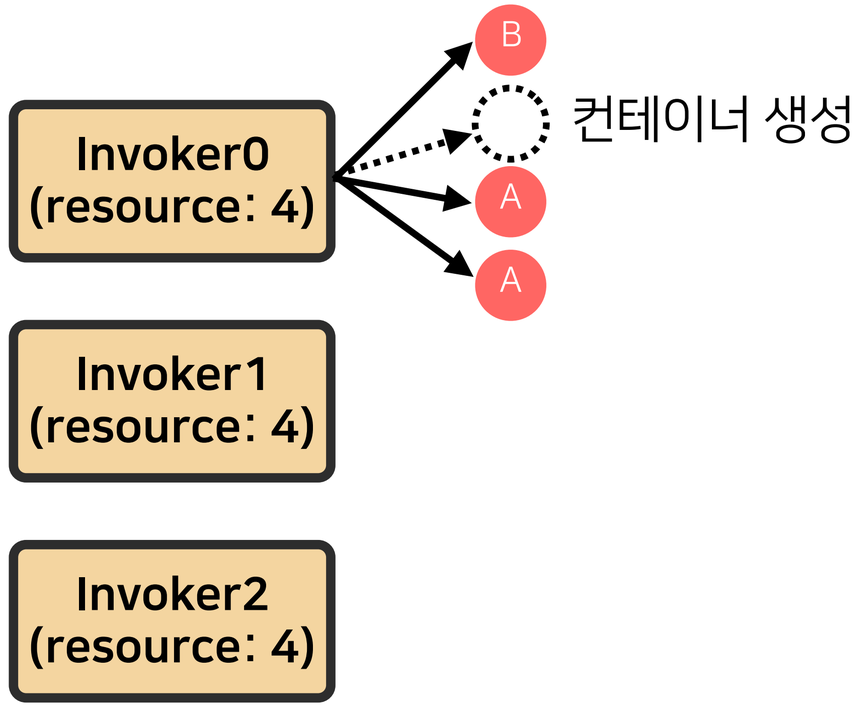
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



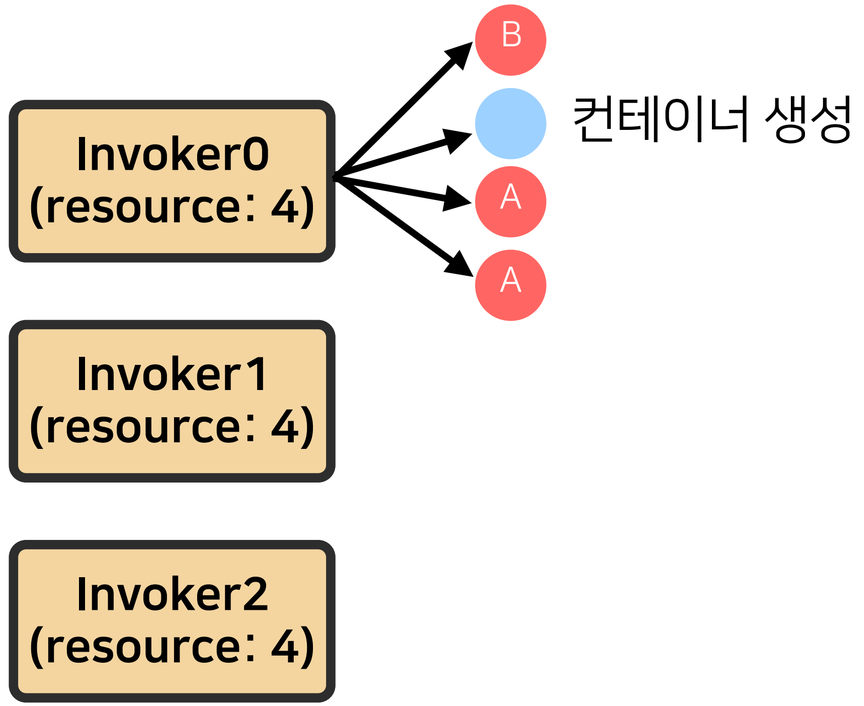
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



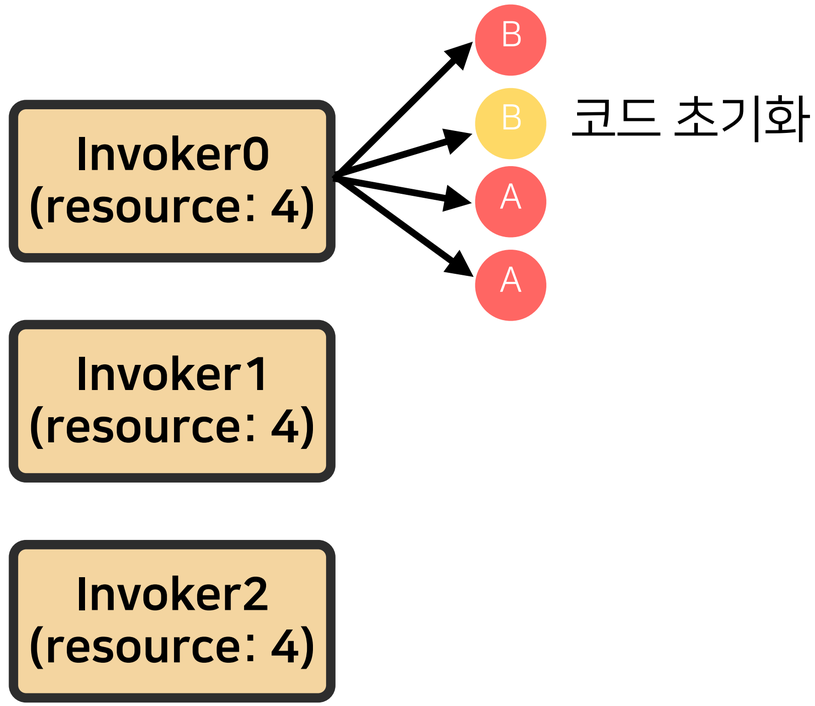
1. 액션간의 간섭

Action A

hash(A) = 0

Action B

hash(B) = 0



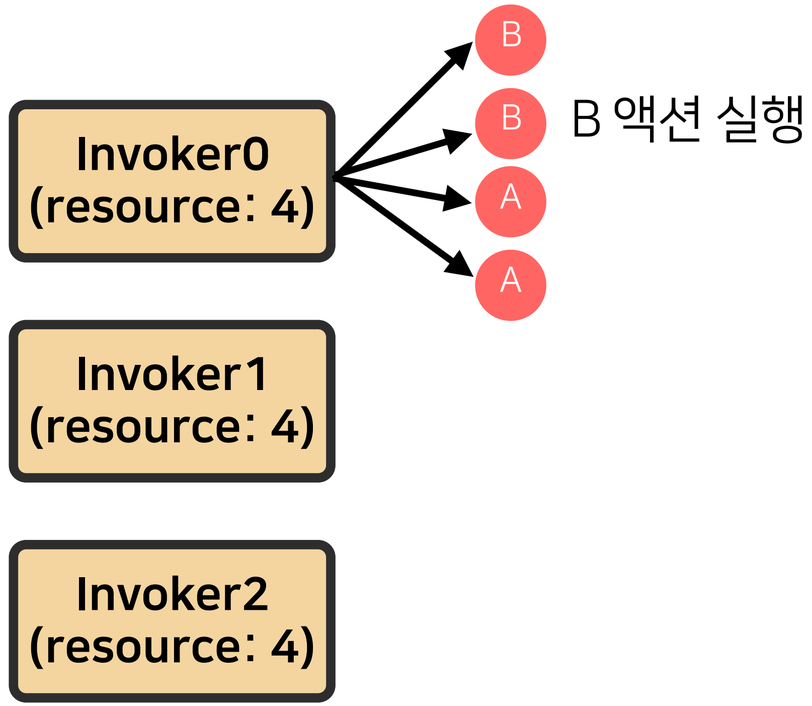
1. 액션간의 간섭

Action A

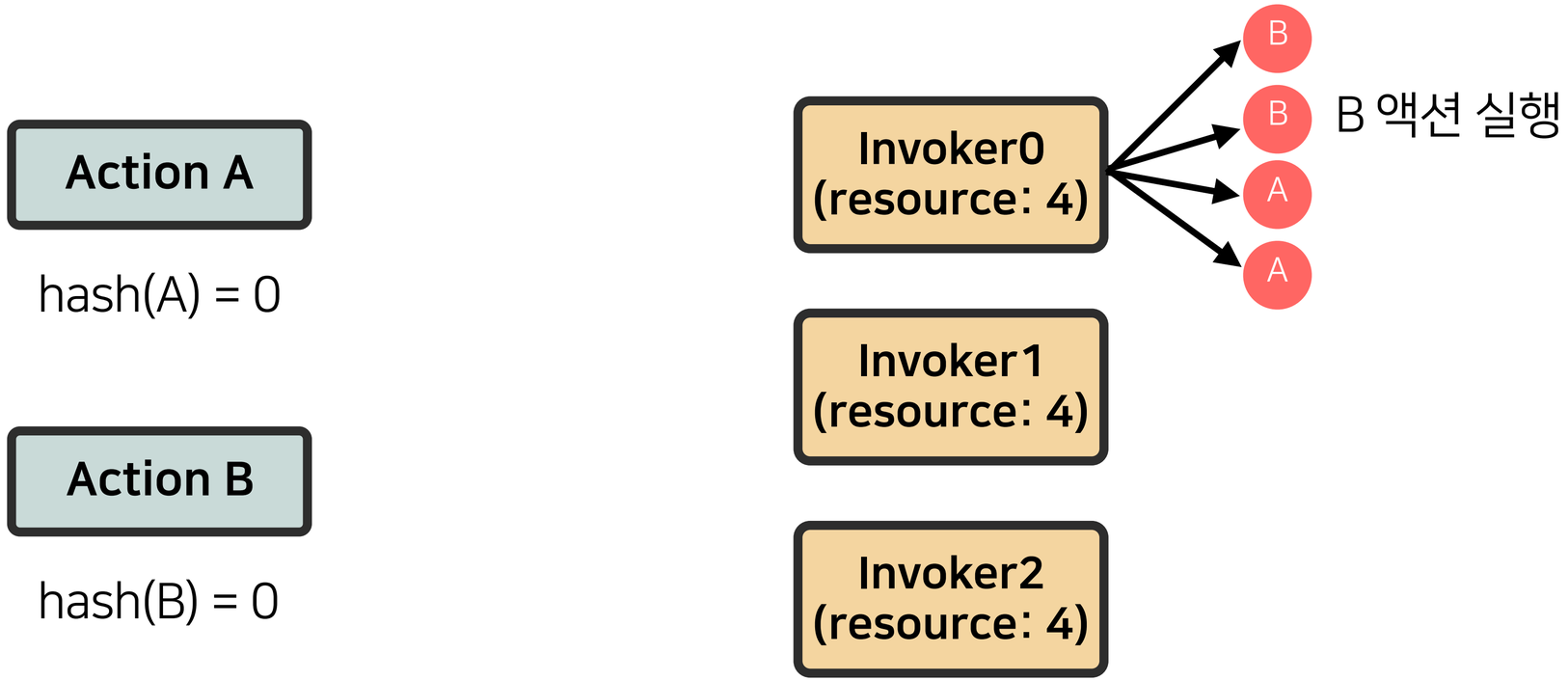
hash(A) = 0

Action B

hash(B) = 0



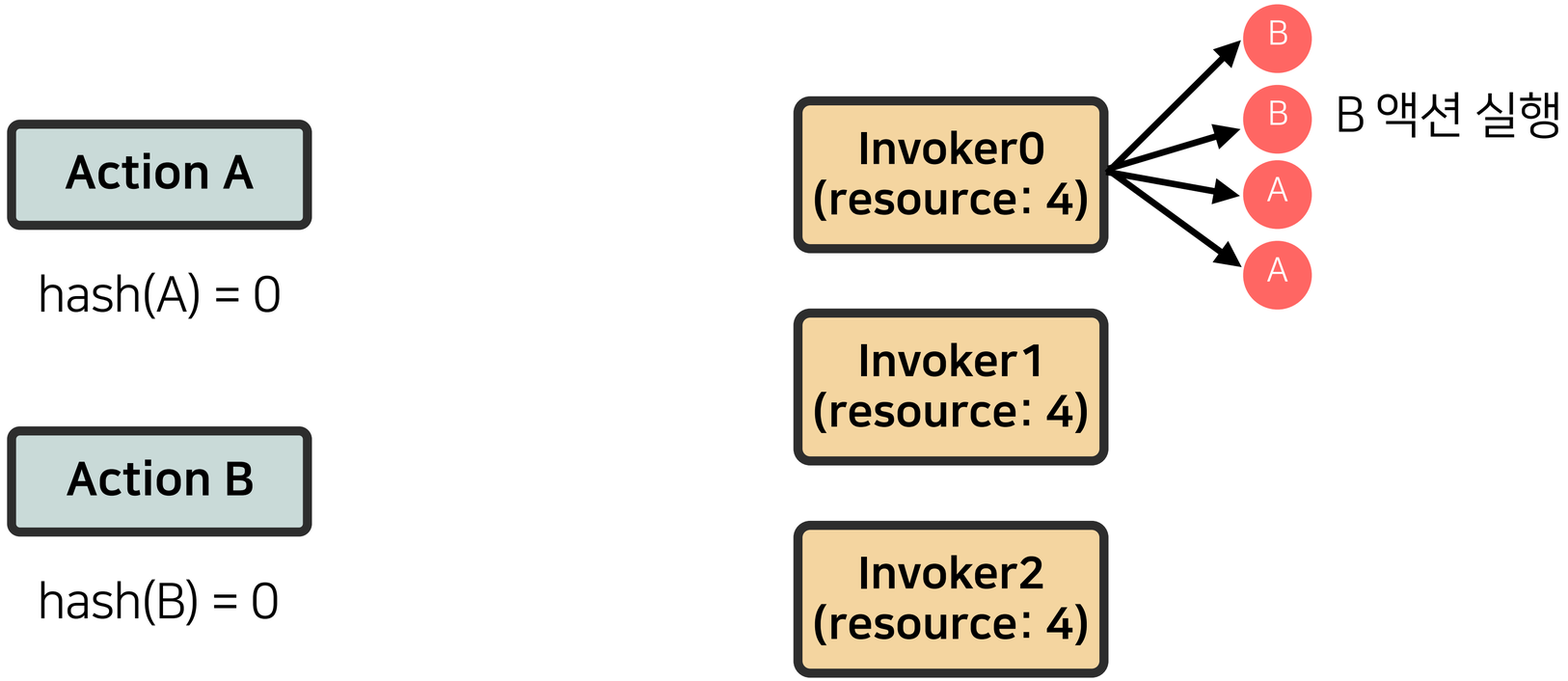
1. 액션간의 간섭



액션 실행시마다 컨테이너의 삭제와 생성이 발생

1. 액션간의 간섭

DEVIEW
2019



다른 Invoker들은 idle한 상태

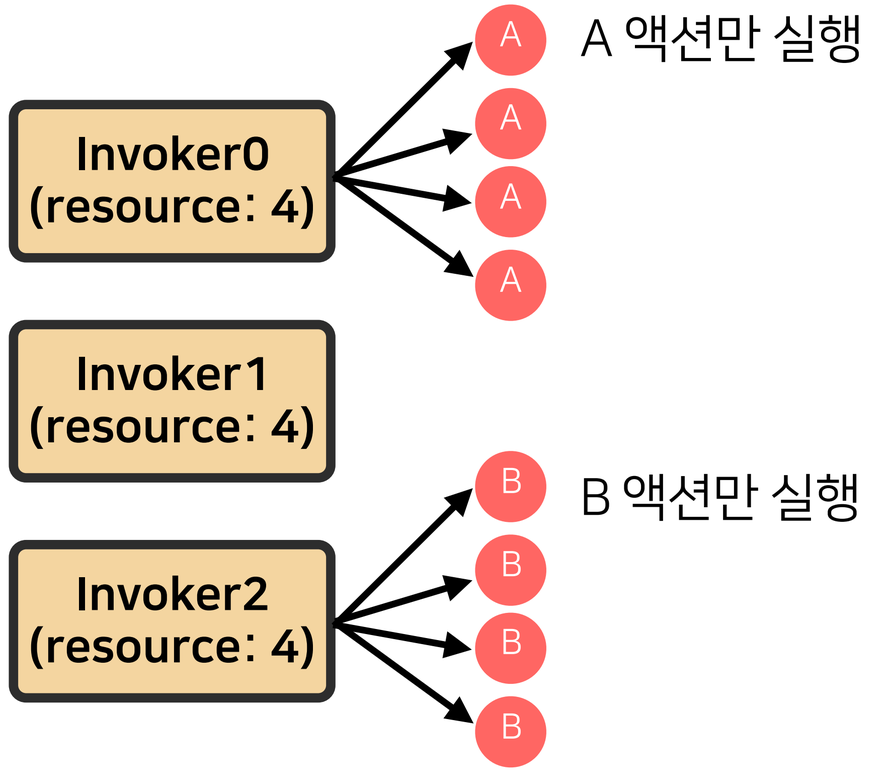
1. 액션간의 간섭 - 이상적인 케이스

Action A

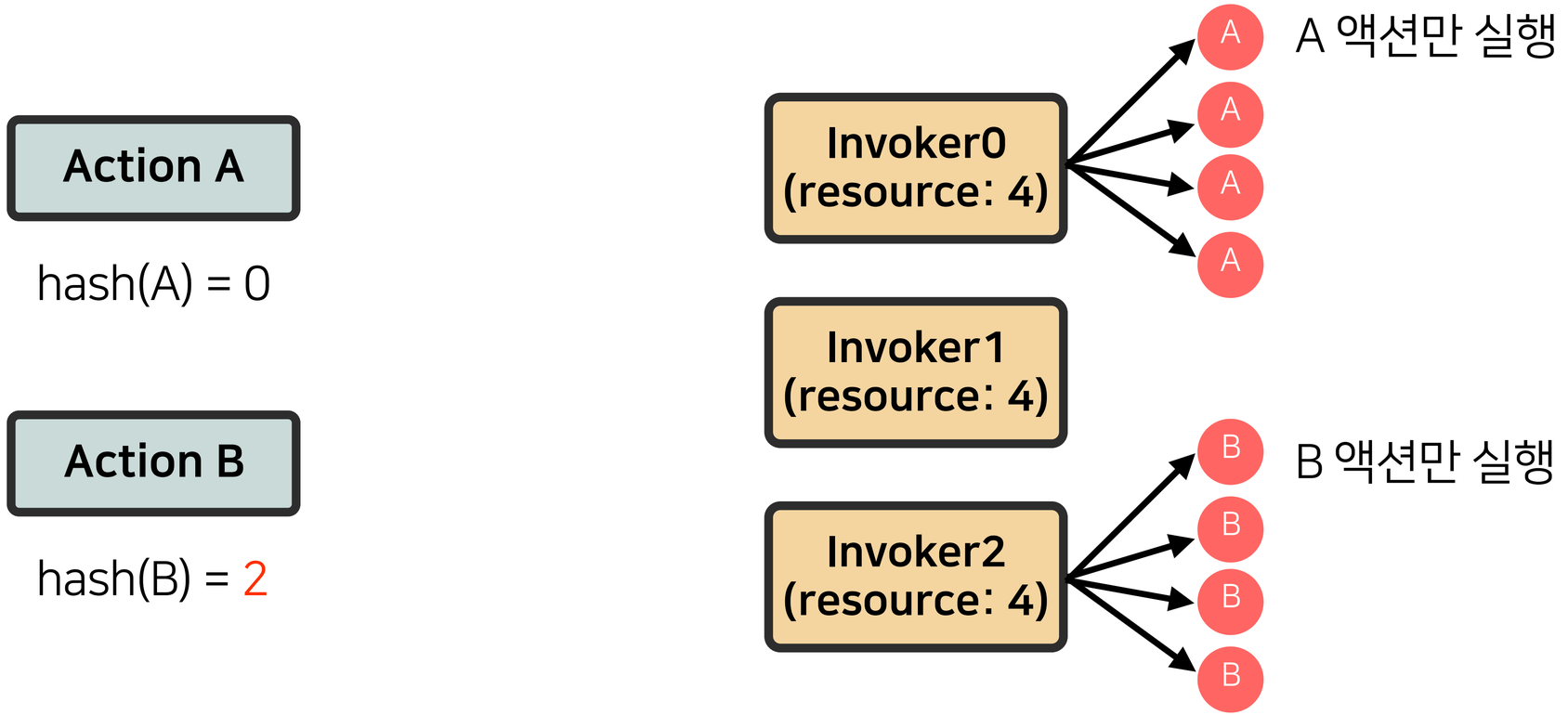
hash(A) = 0

Action B

hash(B) = 2



1. 액션간의 간섭 - 이상적인 케이스



컨테이너의 삭제/생성이 발생하지 않음

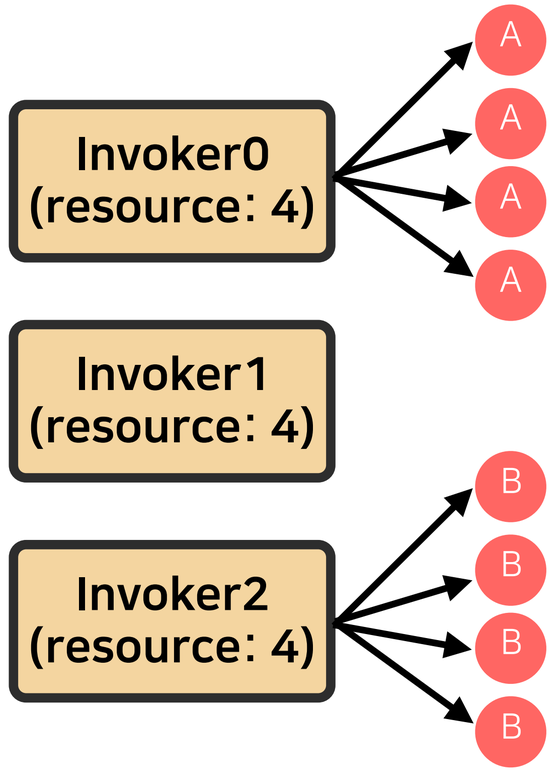
1. 액션간의 간섭 - 이상적인 케이스

Action A

hash(A) = 0

Action B

hash(B) = 2



컨테이너가 100% 재사용 됨

1. 액션간의 간섭

액션의 실행시간: 2ms

1. 액션간의 간섭

액션의 실행시간: 2ms

컨테이너 생성 삭제: 500ms ~ 1300ms

1. 액션간의 간섭

액션의 실행시간: 2ms

컨테이너 생성 삭제: 500ms ~ 1300ms

실제 실행시간: 2ms + 1300ms = 1302ms

1. 액션간의 간섭

액션간의 간섭으로 인해

1. 액션간의 간섭

액션간의 간섭으로 인해

최대 650배 가량 느려짐

1. 액션간의 간섭

액션간의 간섭으로 인해

최대 650배 가량 느려짐

이후의 모든 실행도 함께 느려짐

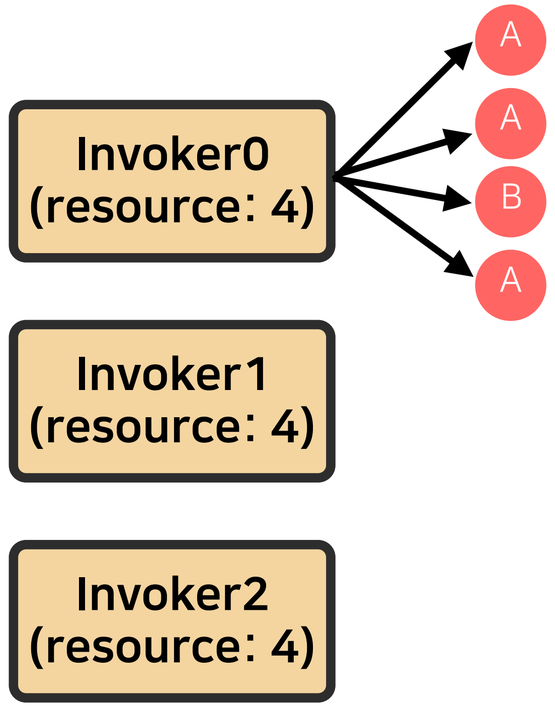
2. 앞선 실행을 기다리지 않음

Action A

hash(A) = 0

Action B

hash(B) = 0



2. 앞선 실행을 기다리지 않음

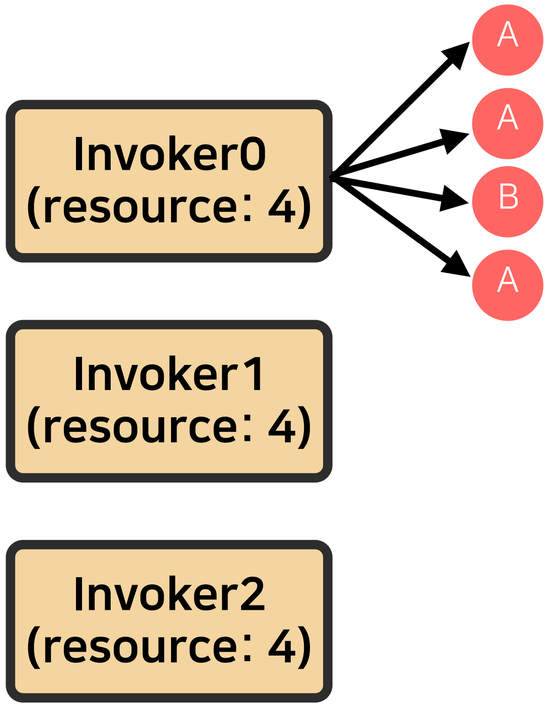
Action A

hash(A) = 0

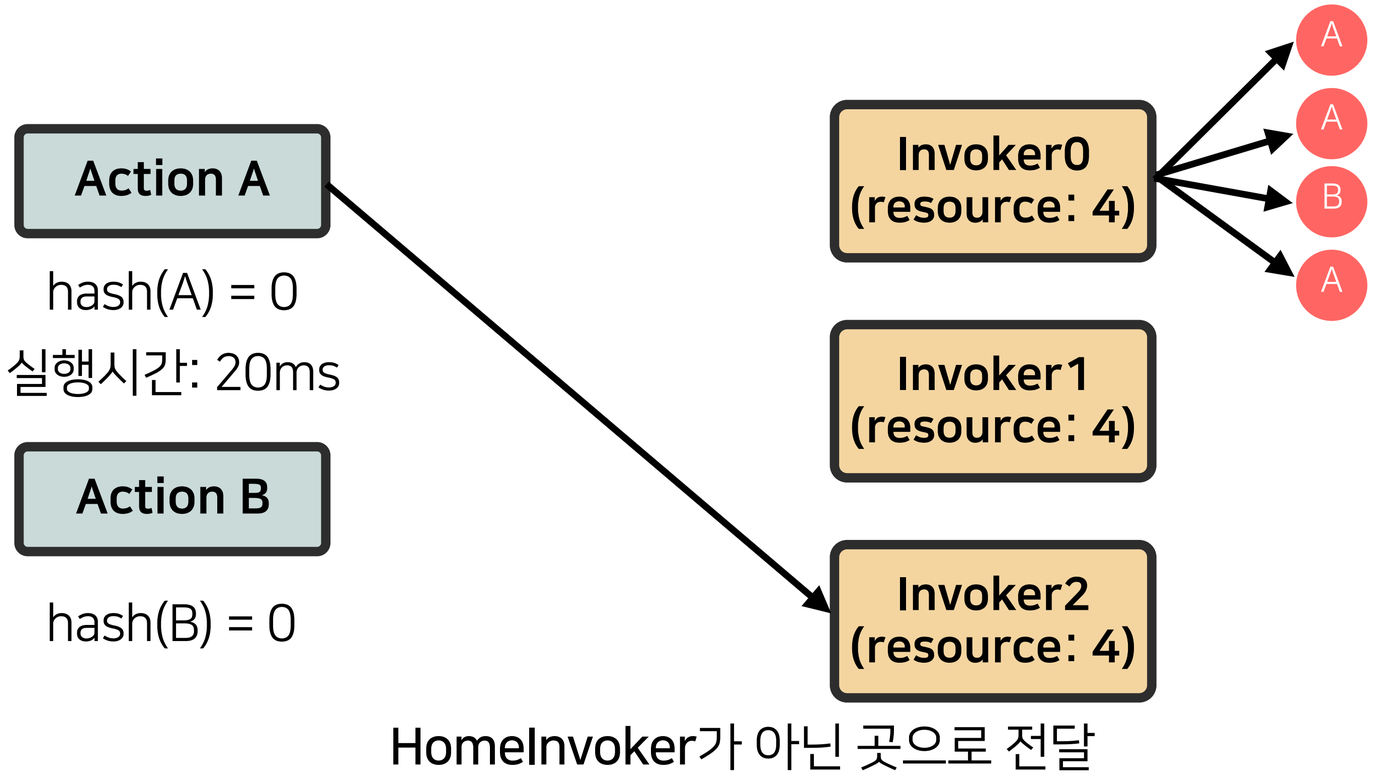
실행시간: 20ms

Action B

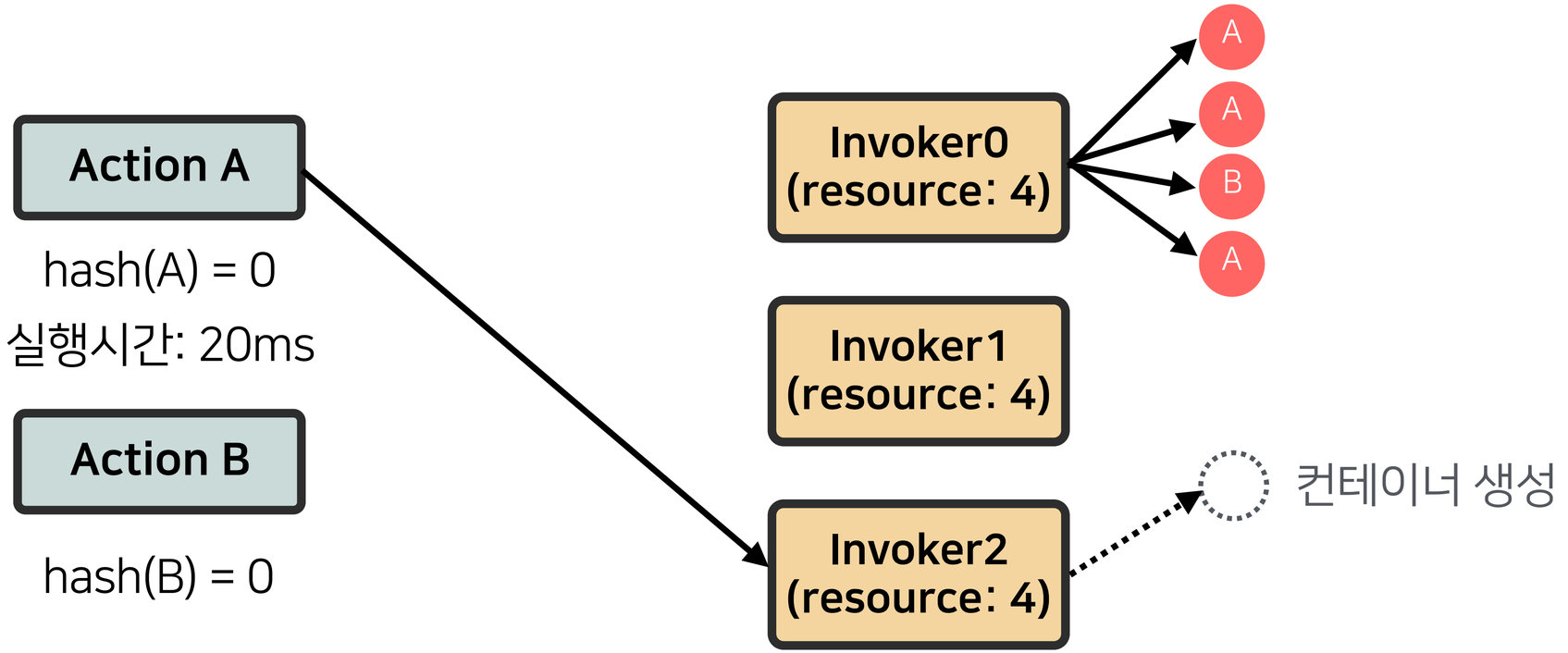
hash(B) = 0



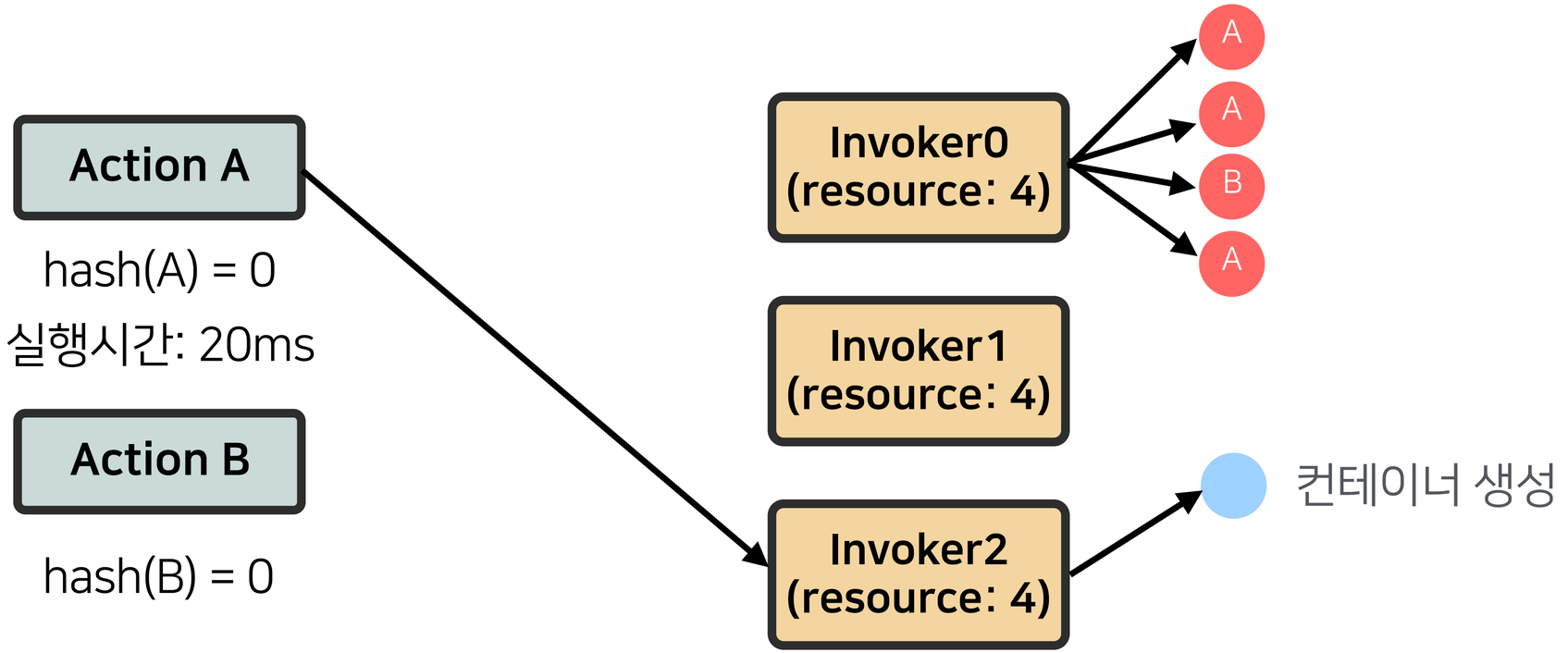
2. 앞선 실행을 기다리지 않음



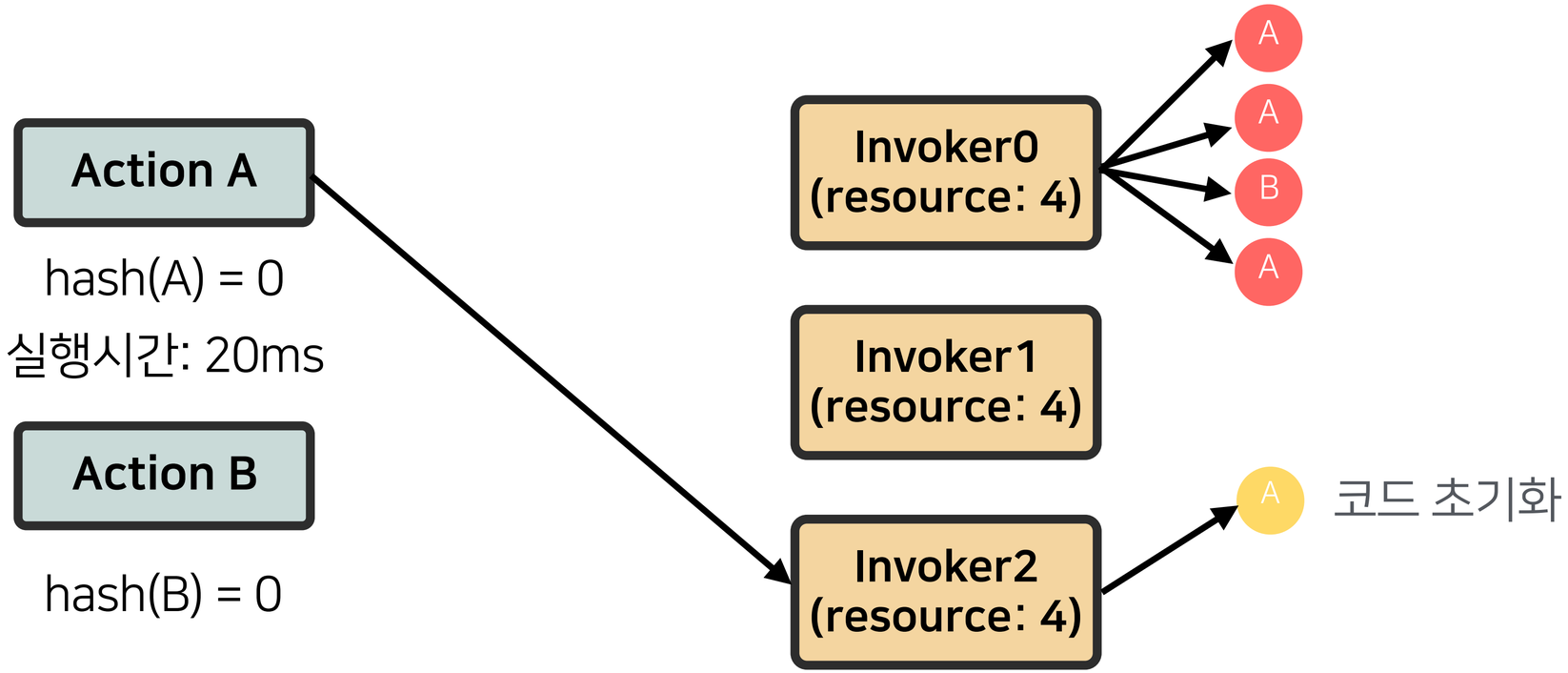
2. 앞선 실행을 기다리지 않음



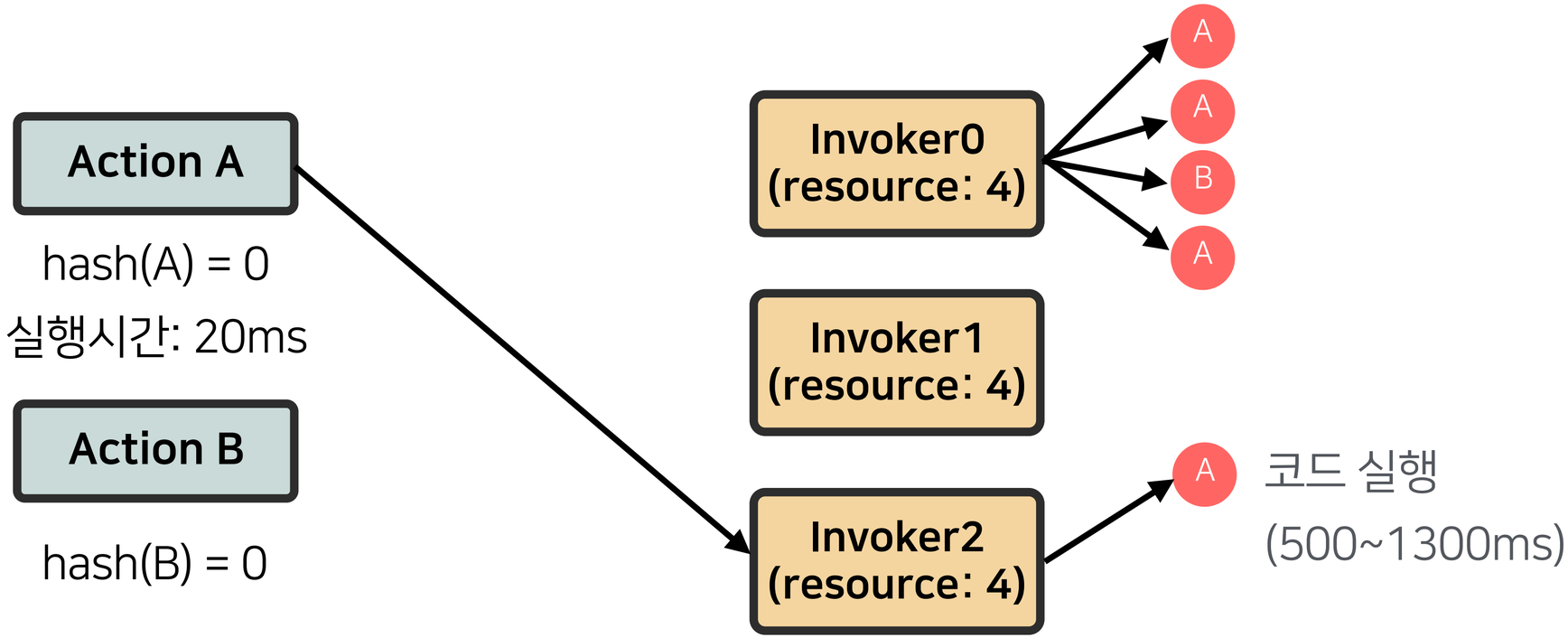
2. 앞선 실행을 기다리지 않음



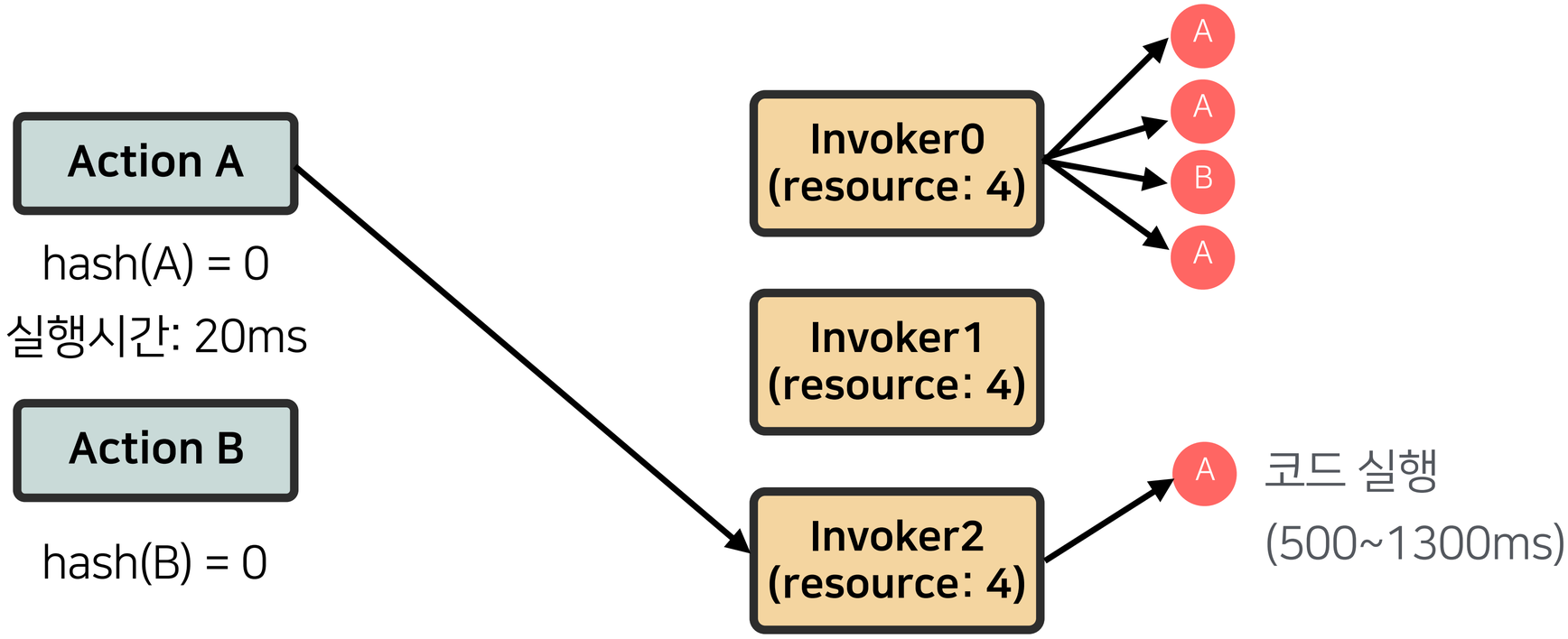
2. 앞선 실행을 기다리지 않음



2. 앞선 실행을 기다리지 않음



2. 앞선 실행을 기다리지 않음



500ms + 20ms = 520ms

2. 앞선 실행을 기다리지 않음 - 이상적인 경우

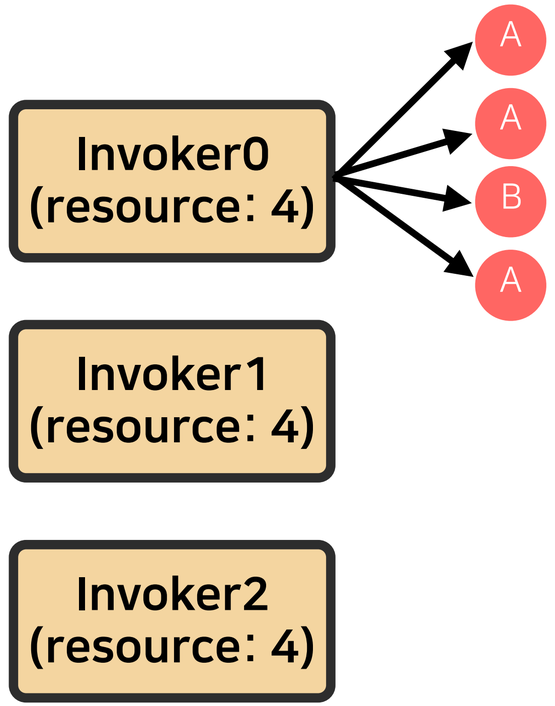
Action A

hash(A) = 0

실행시간: 20ms

Action B

hash(B) = 0



2. 앞선 실행을 기다리지 않음 - 이상적인 경우

스케쥴링을 하지 않고 기다림

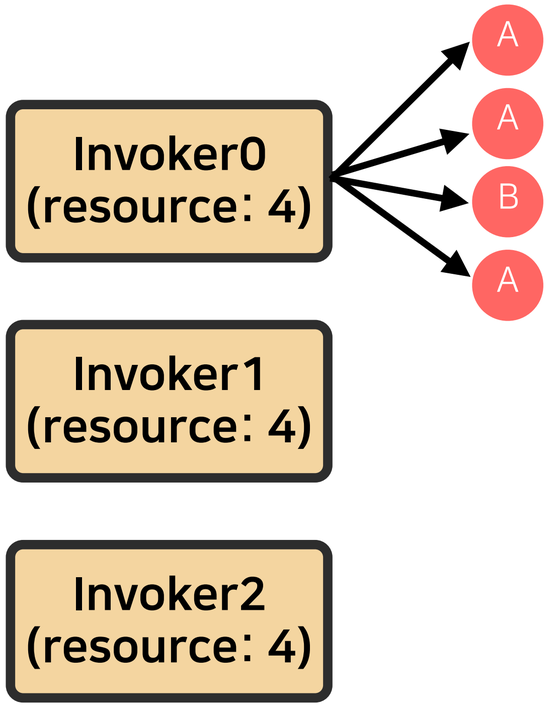
Action A

hash(A) = 0

실행시간: 20ms

Action B

hash(B) = 0



2. 앞선 실행을 기다리지 않음 - 이상적인 경우

DEVIEW 2019

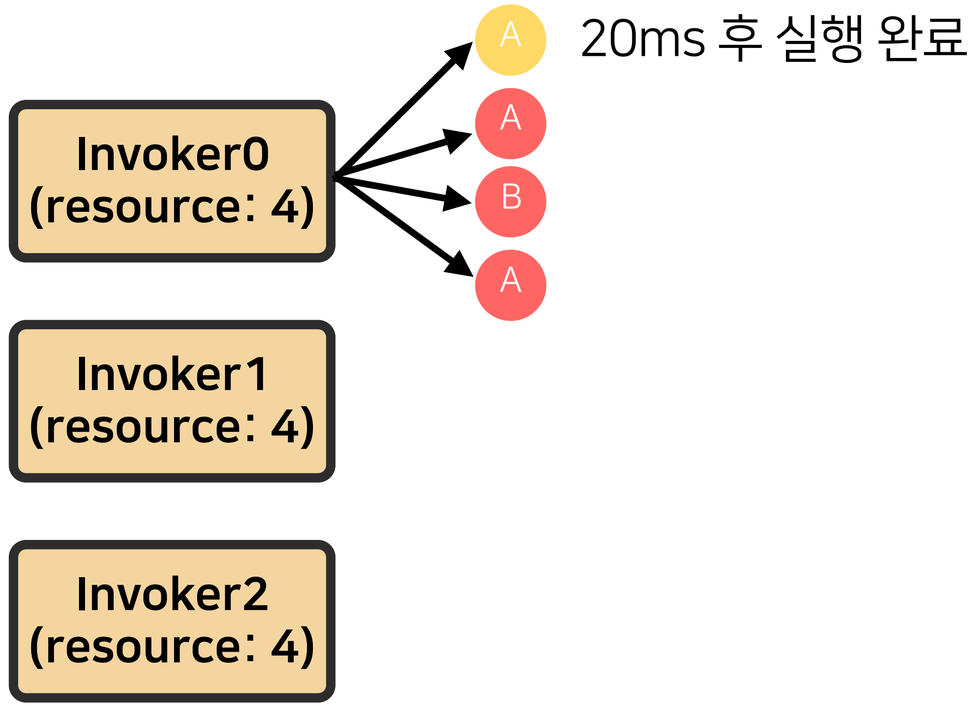
Action A

hash(A) = 0

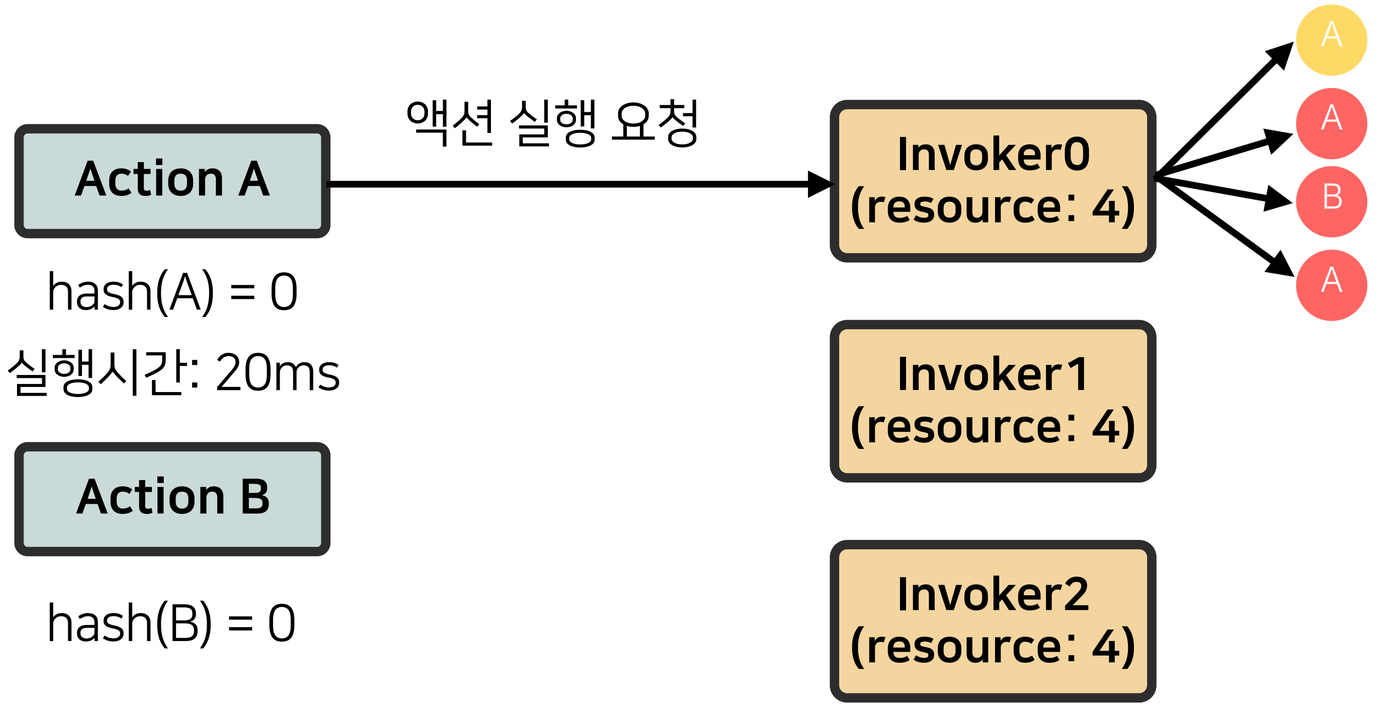
실행시간: 20ms

Action B

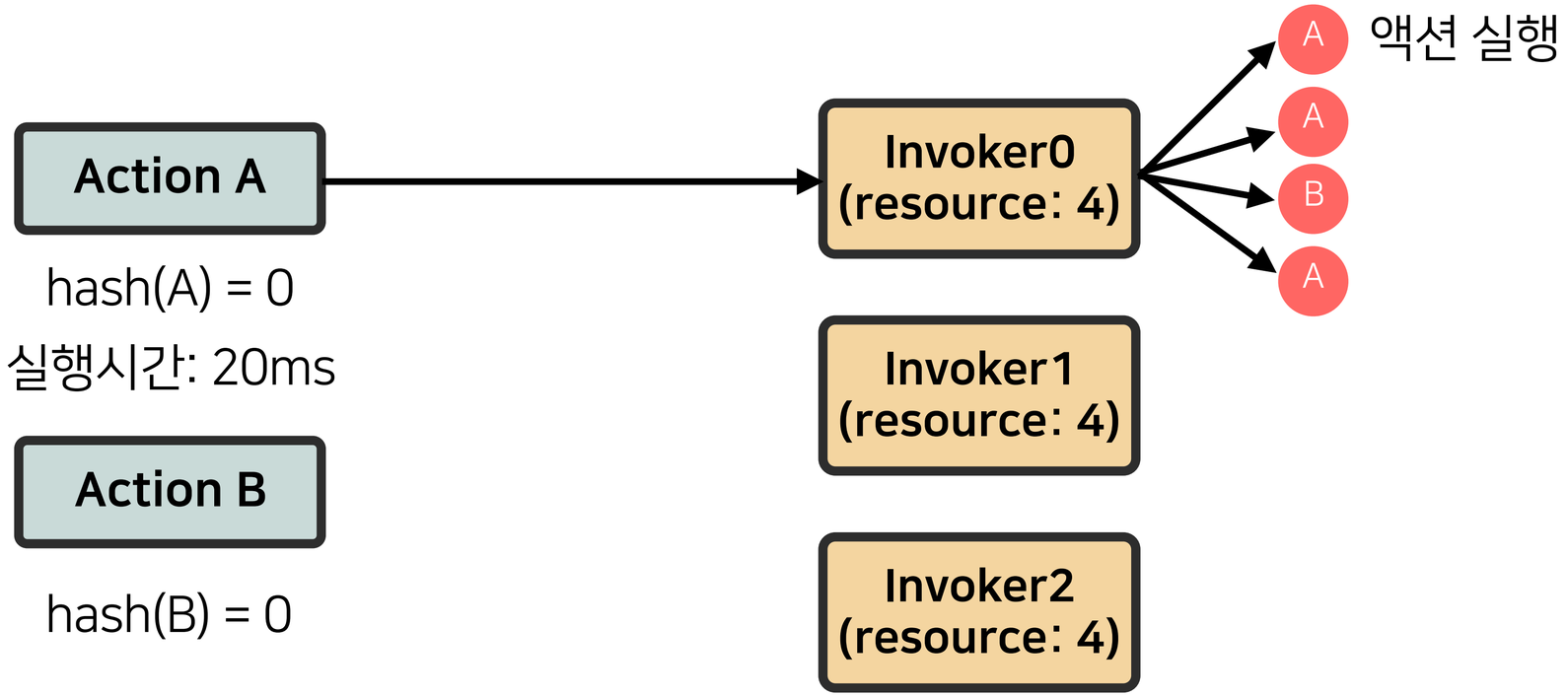
hash(B) = 0



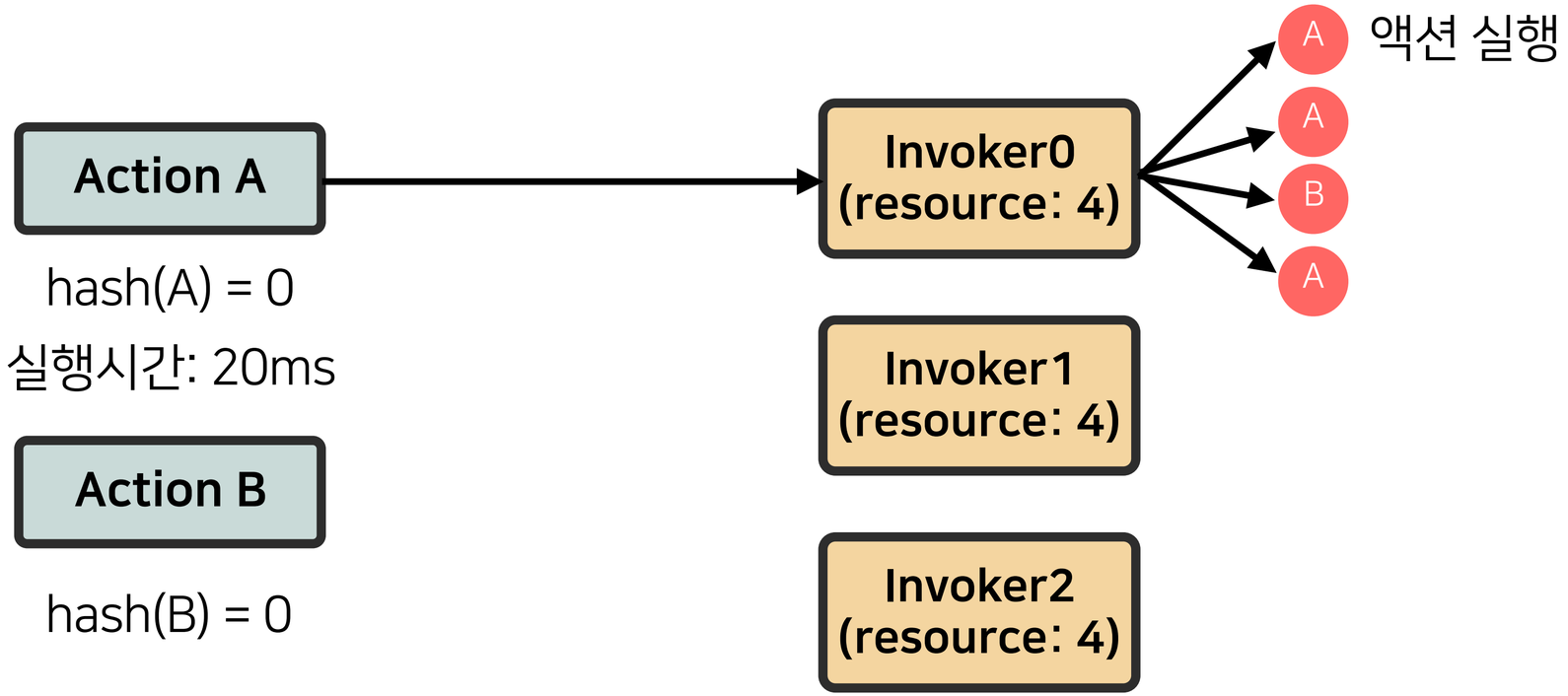
2. 앞선 실행을 기다리지 않음 - 이상적인 경우



2. 앞선 실행을 기다리지 않음 - 이상적인 경우



2. 앞선 실행을 기다리지 않음 - 이상적인 경우



20ms + 20ms = 40ms

2. 앞선 실행을 기다리지 않음

DEVIEW
2019

520ms vs 40ms

2. 앞선 실행을 기다리지 않음

DEVIEW
2019

520ms vs 40ms

13배 더 느림

2. 앞선 실행을 기다리지 않음

실행시간 <= 500ms 인 경우,

ColdStart 보다 이전 실행을 기다리는 것이 낫다

2. 앞선 실행을 기다리지 않음

DEVIEW
2019

실행시간 <= 500ms 인 경우,

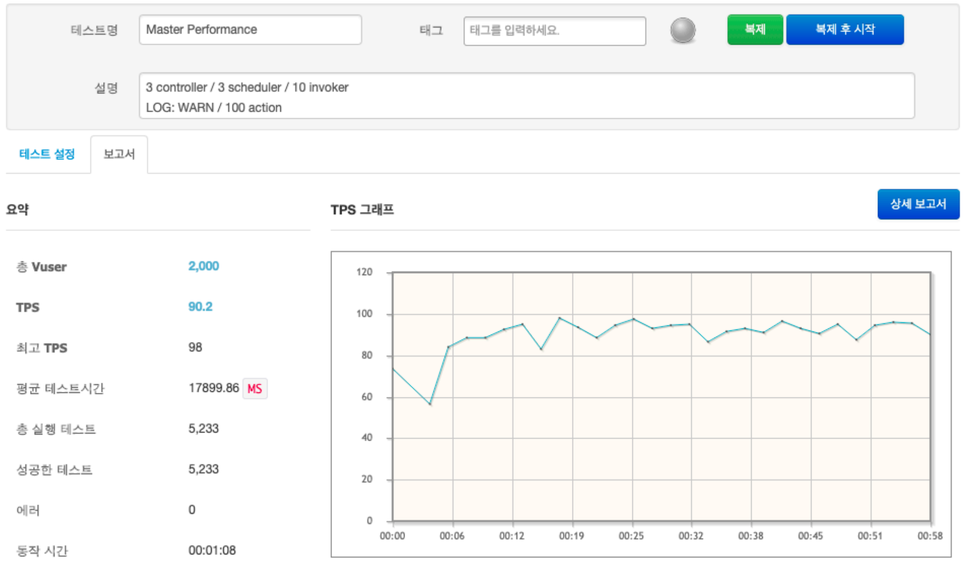
ColdStart 보다 이전 실행을 기다리는 것이 낫다

https://cwiki.apache.org/confluence/display/OPENWHISK/Autonomous+Container+Scheduling+v1

Docker의 성능이 OpenWhisk에 미치는 영향

DEVIEW 2019

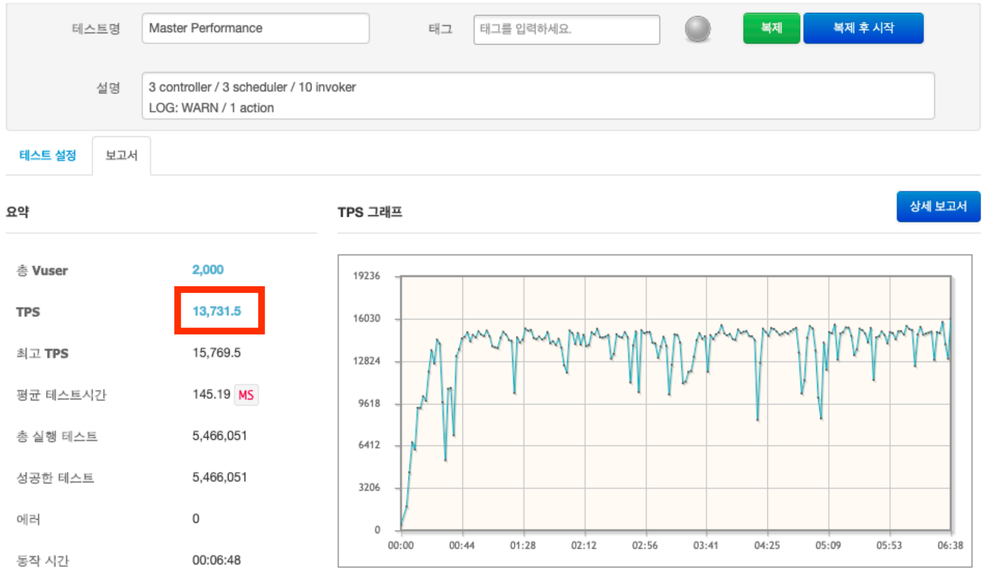
액션 1개



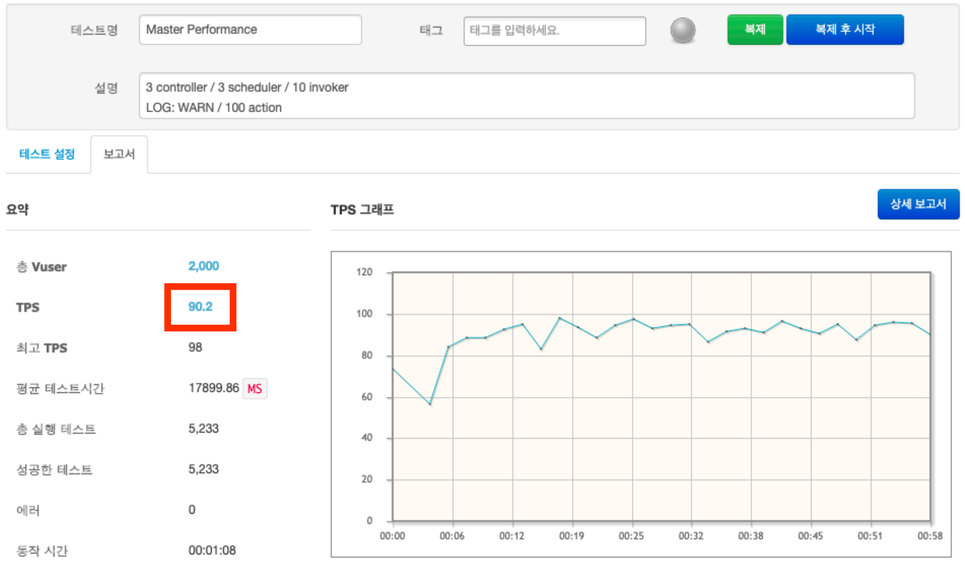
액션 100개

Docker의 성능이 OpenWhisk에 미치는 영향

DEVIEW 2019



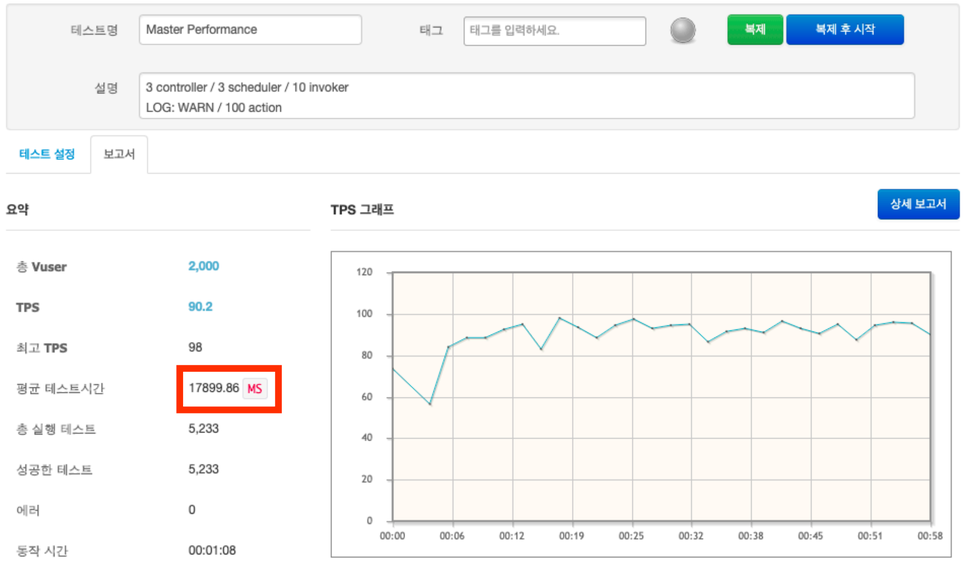
액션 1개: 14,000 TPS



액션 100개: 90 TPS

Docker의 성능이 OpenWhisk에 미치는 영향

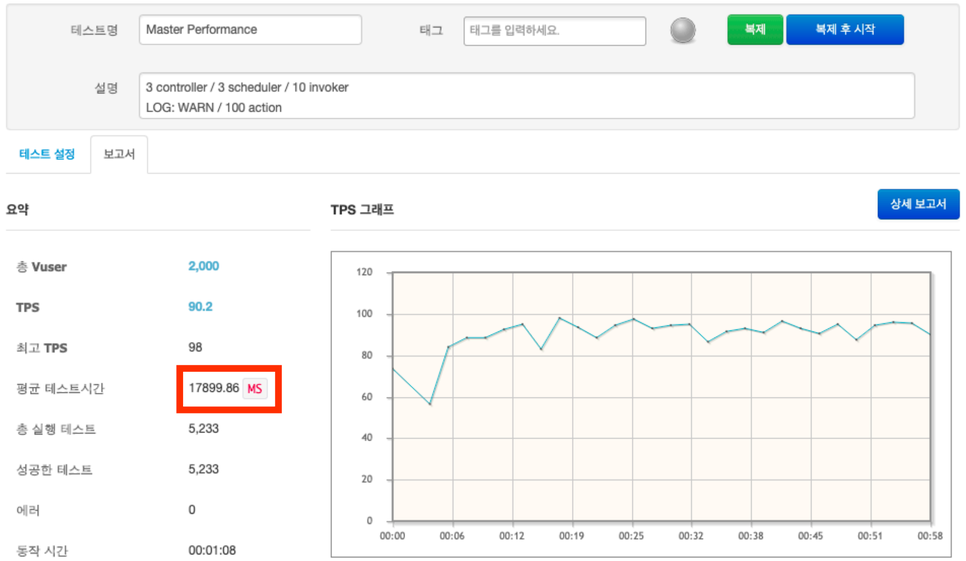
DEVIEW 2019



액션간의 간섭으로 인해 실행시간이 평균 17초까지 늘어남

Docker의 성능이 OpenWhisk에 미치는 영향

DEVIEW 2019



TPS가 약 155배 가량 줄어듬

시스템의 성능 결정 불가

DEVIEW
2019

TPS: 20,000

100명의 유저가 1개 액션 사용

시스템의 성능 결정 불가

TPS: 20,000

100명의 유저가 1개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

시스템의 성능 결정 불가

DEVIEW 2019

TPS: 20,000

100명의 유저가 1개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 30

100명의 유저가 100개 액션 사용

시스템의 성능 결정 불가

DEVIEW 2019

TPS: 20,000

100명의 유저가 1개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 30

100명의 유저가 100개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

시스템의 성능 결정 불가

DEVIEW 2019

TPS: 20,000

100명의 유저가 1개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 30

100명의 유저가 100개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 2,400

200명의 유저가 10개 액션 사용

시스템의 성능 결정 불가

DEVIEW 2019

TPS: 20,000

100명의 유저가 1개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 30

100명의 유저가 100개 액션 사용

TPS: 6,000

100명의 유저가 10개 액션 사용

TPS: 2,400

200명의 유저가 10개 액션 사용

TPS: 30

750명의 유저가 100개 액션 사용

시스템의 성능 결정 불가

동일한 환경에서도

실행되는 액션의 수와

실행하는 유저의 수에 따라

성능이 달라짐

시스템의 성능 결정 불가

언제 Scale-out할지,

얼마나 서버를 늘릴지 결정 불가

Resource Planning 불가

DEVIEW
2019

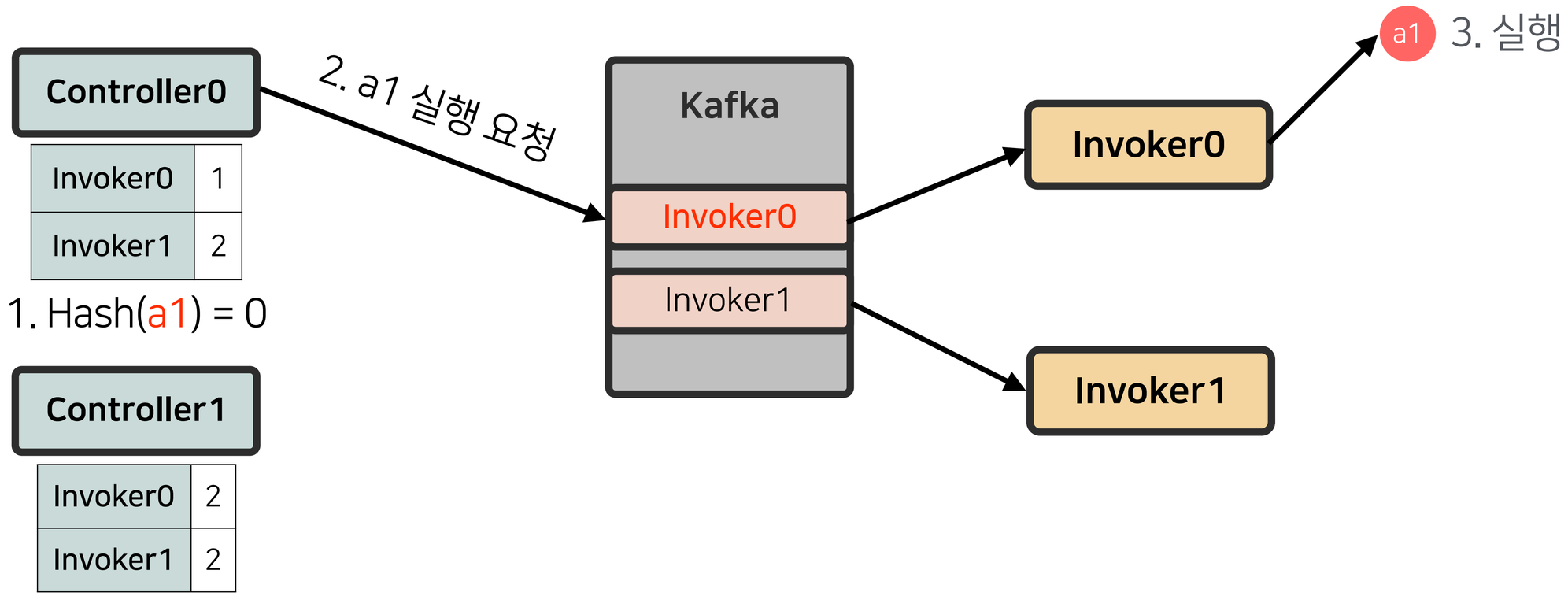
성능 이슈 극복하기

성능 이슈 극복하기

DEVIEW
2019

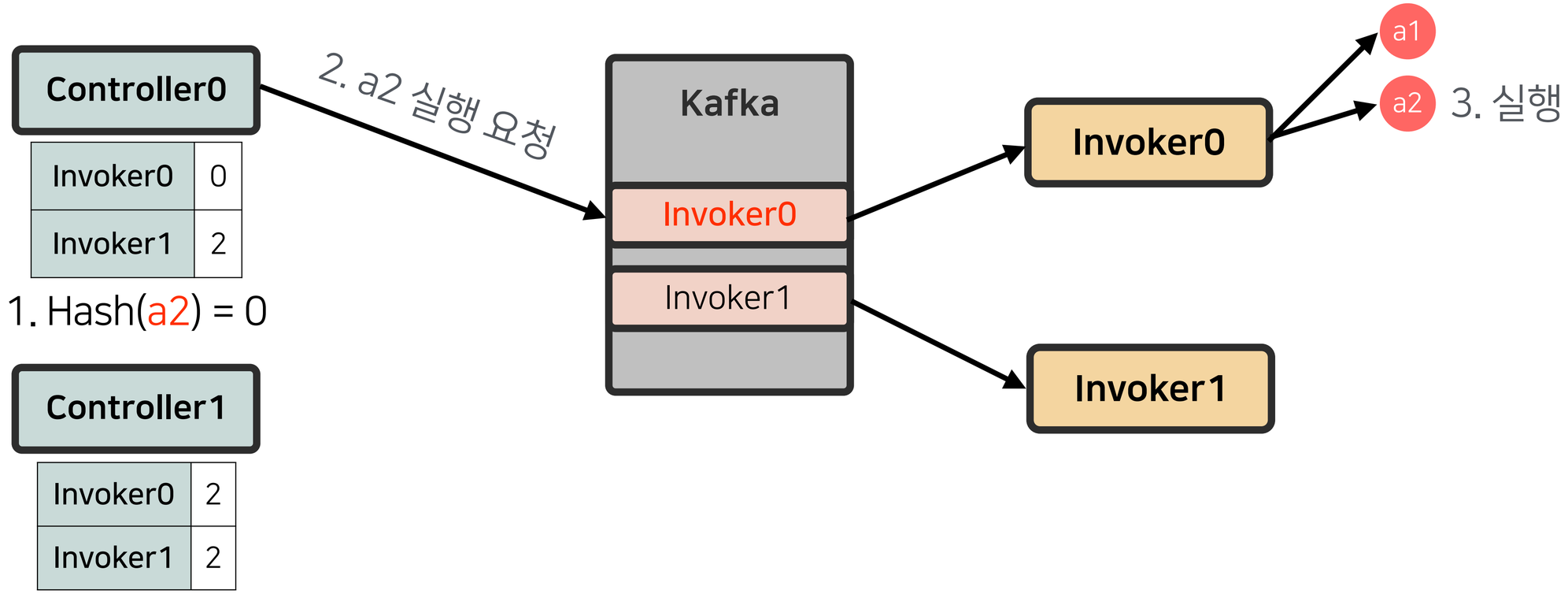
- 액션 별로 별도의 큐를 사용
- 액션의 실행과 컨테이너의 생성을 분리
- 신규 컴포넌트 도입

기존 방식 - 액션간 간섭

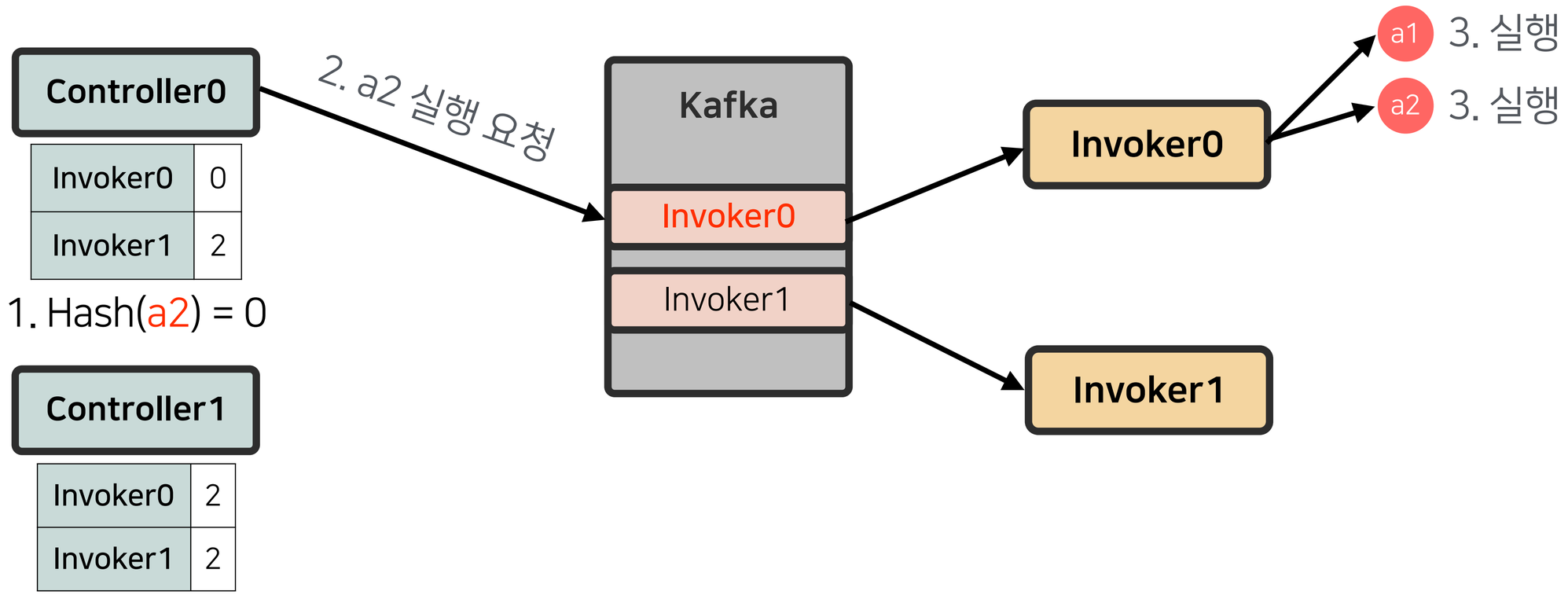

Controller0
Invoker0 1
Invoker1 2
1. Hash(a1) = 0
2. a1 실행 요청
Kafka
Invoker0
Invoker1
Invoker0
Invoker1
a1
3. 실행
Controller1
Invoker0 2
Invoker1 2

기존 방식 - 액션간 간섭

DEVIEW 2019

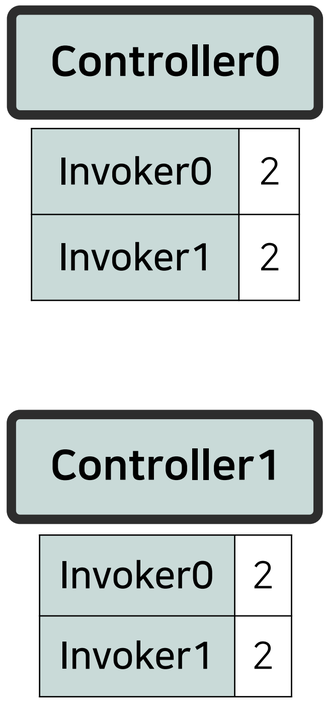


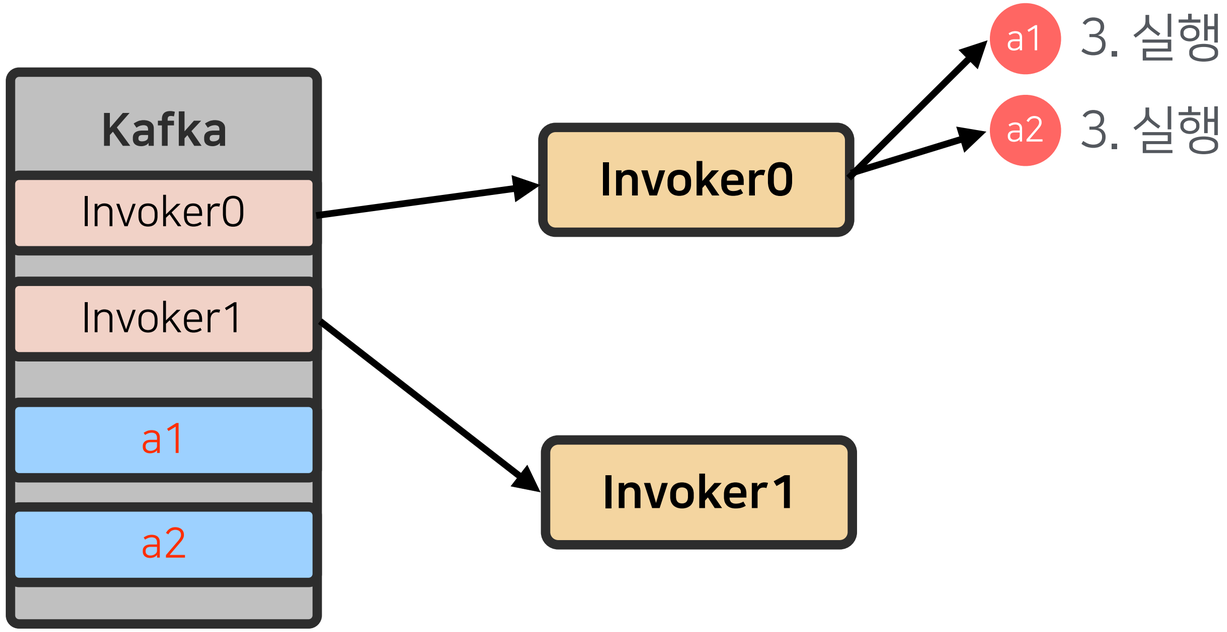
기존 방식 - 액션간 간섭



a1의 실행이 지연되면, a2의 실행도 지연됨

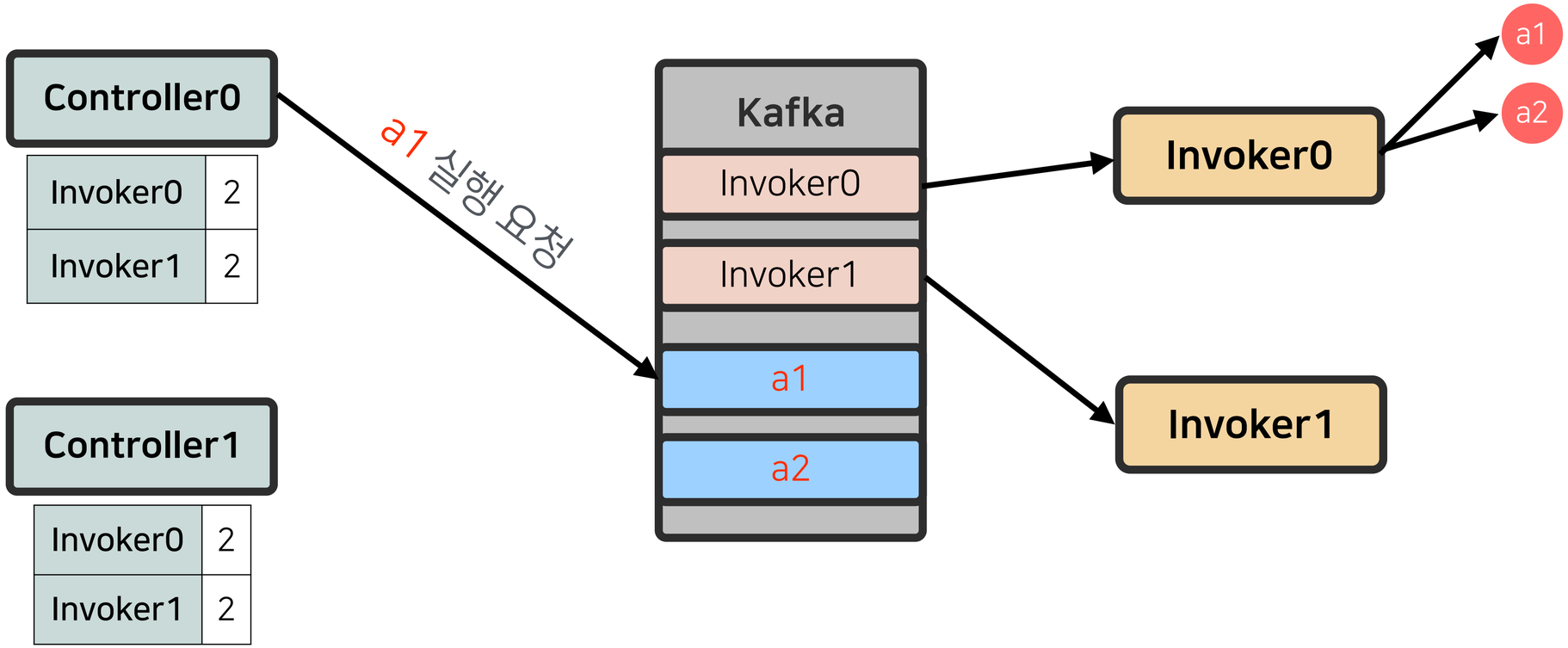
액션 별 메시지 큐 도입


Controller0
Invoker0
2
Invoker1
2
Controller1
Invoker0
2
Invoker1
2


Kafka
Invoker0
Invoker1
a1
a2
Invoker0
Invoker1
a1
3. 실행
a2
3. 실행

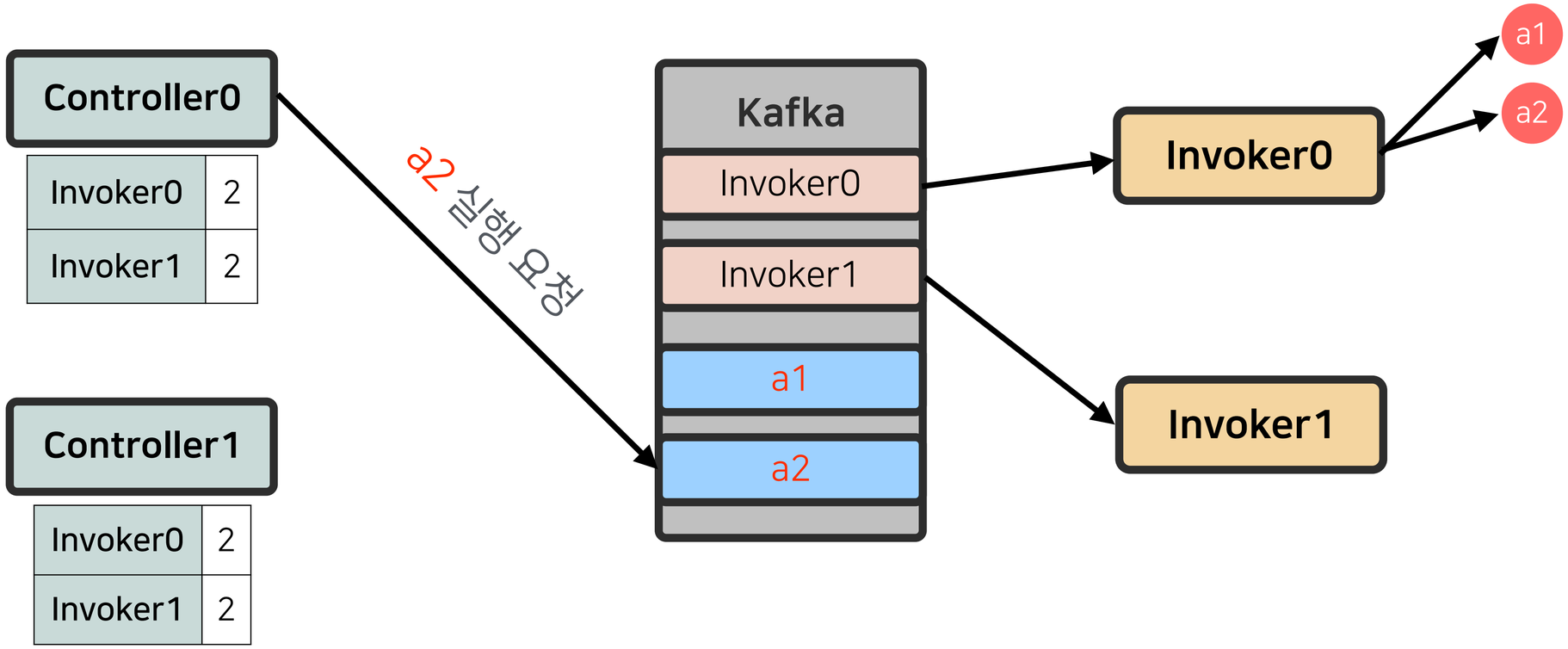
액션 별 메시지 큐 도입

DEVIEW 2019

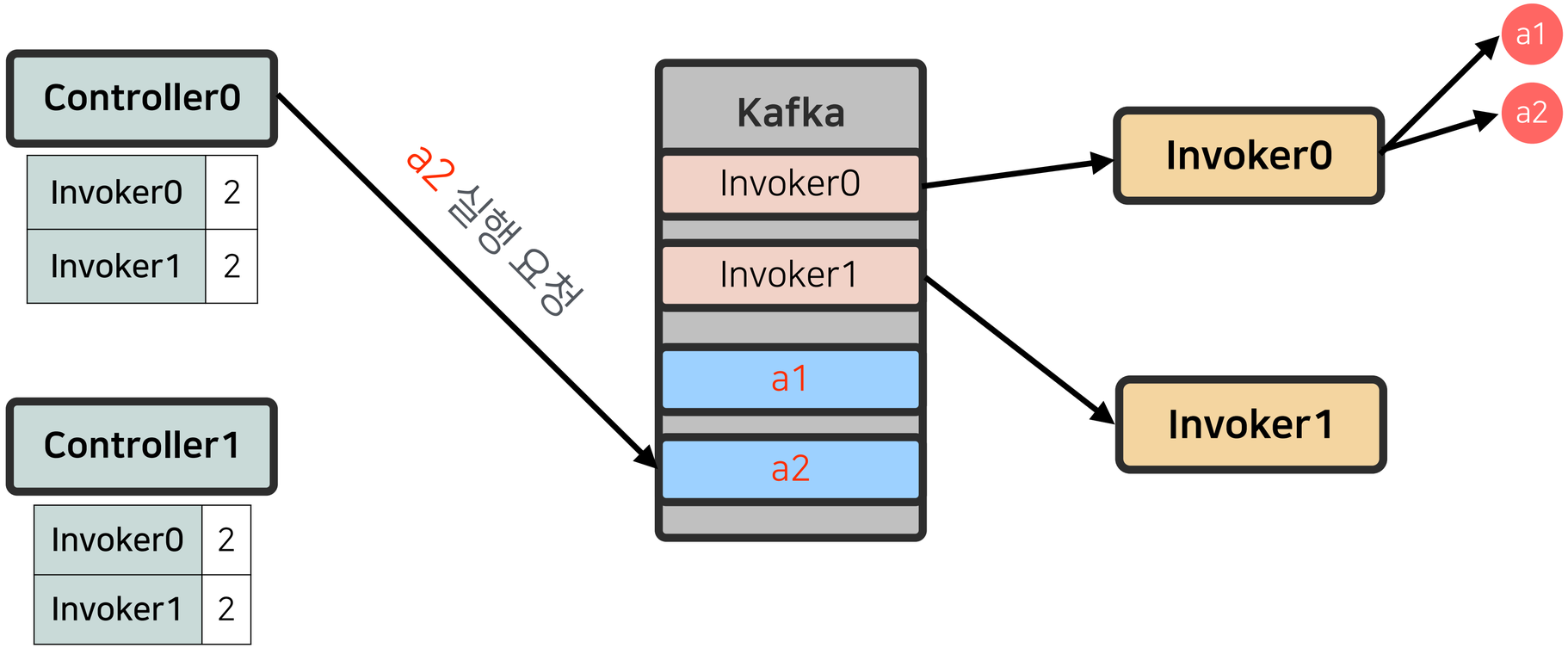


액션 별 메시지 큐 도입

DEVIEW 2019



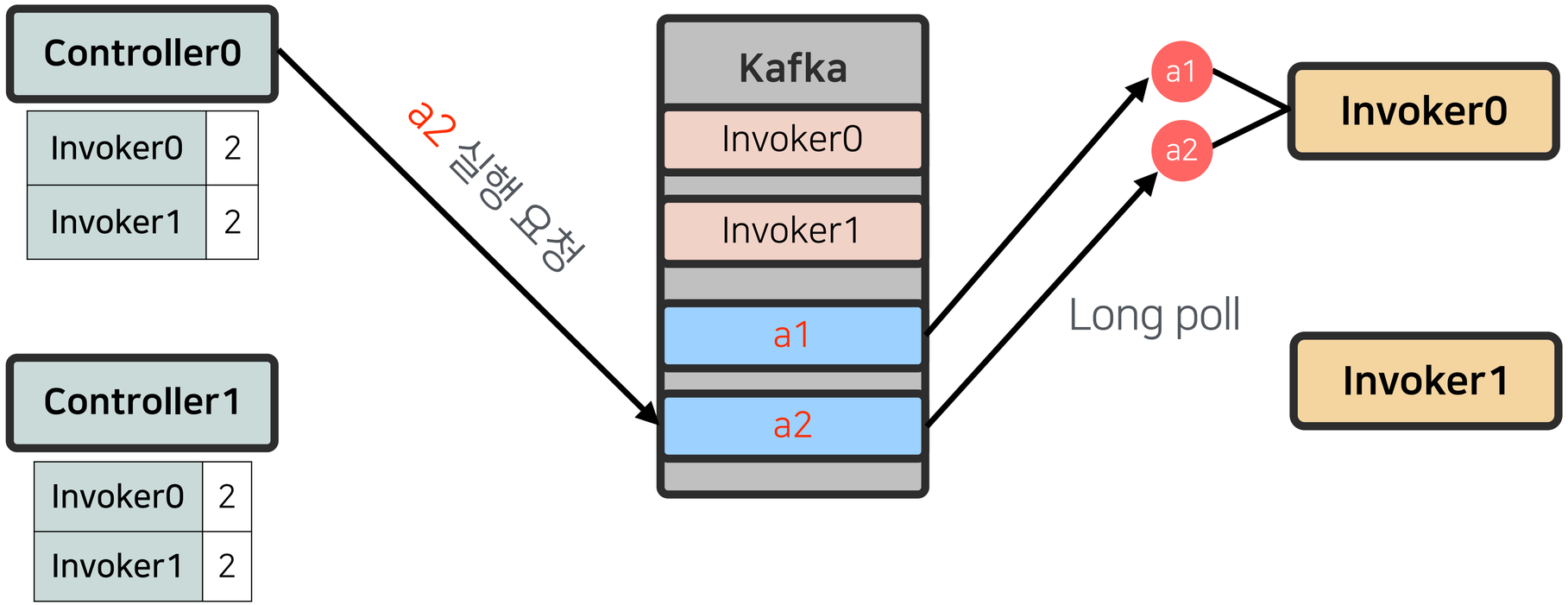
액션 별 메시지 큐 도입



한 액션의 지연이 다른 액션에 영향을 주지 않음

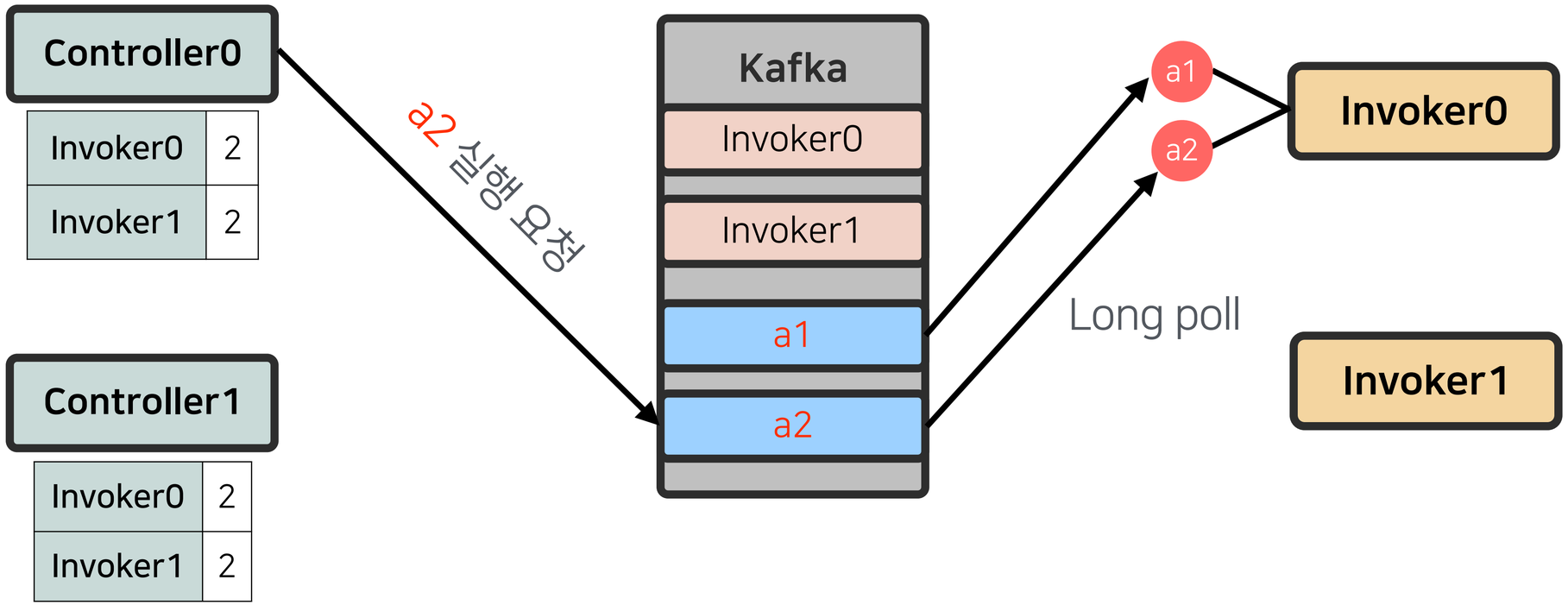
Pull 기반 스케쥴링

DEVIEW 2019



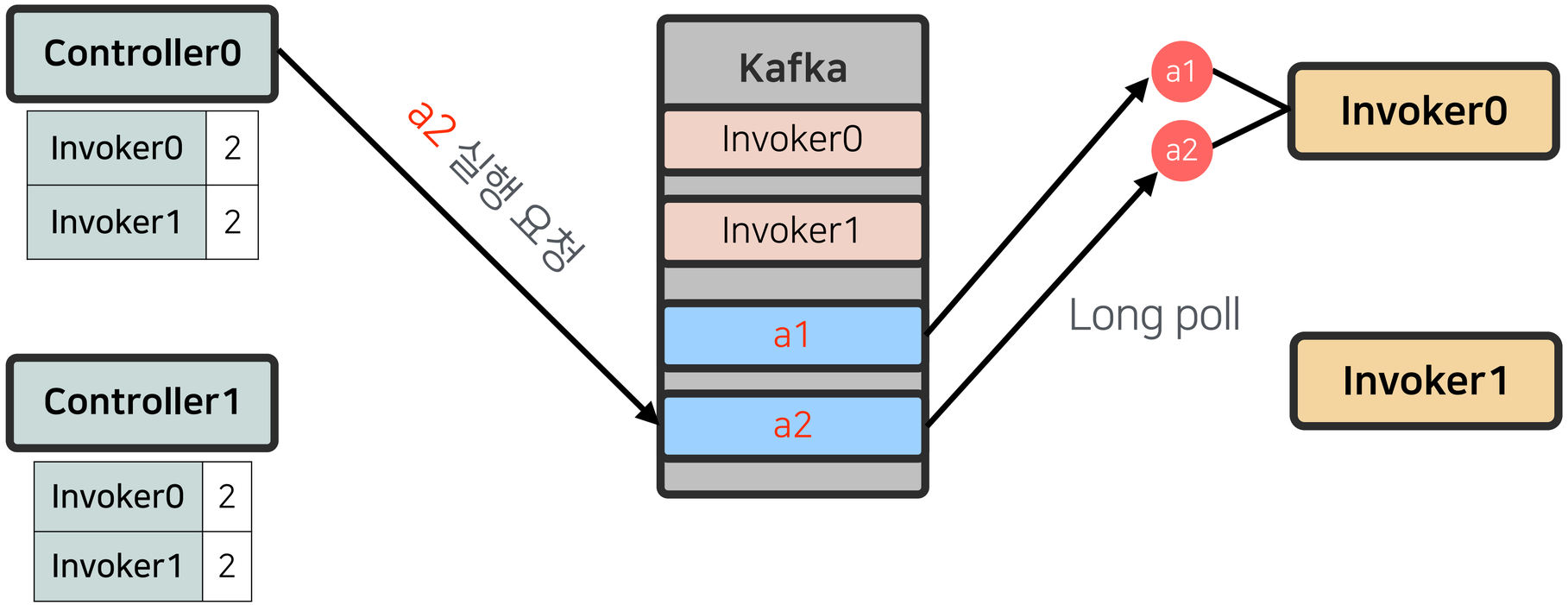
각 컨테이너가 메시지를 직접 가져감

Pull 기반 스케쥴링



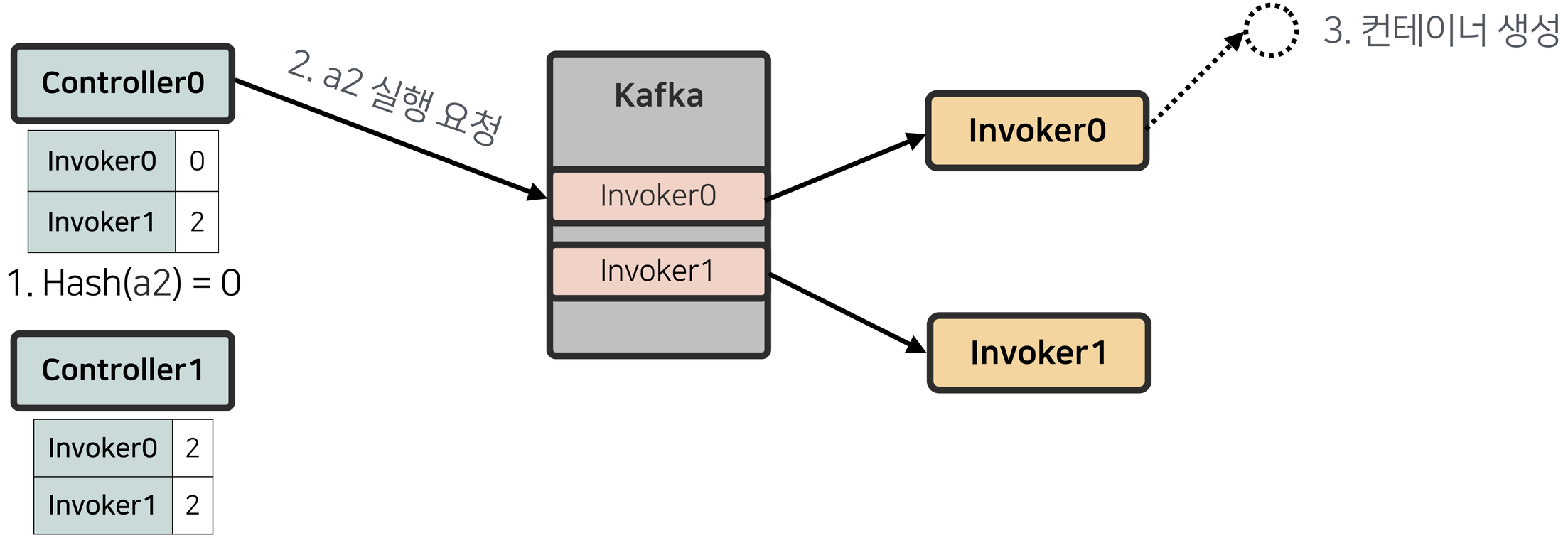
컨테이너의 위치를 신경쓰지 않아도 됨

Pull 기반 스케쥴링

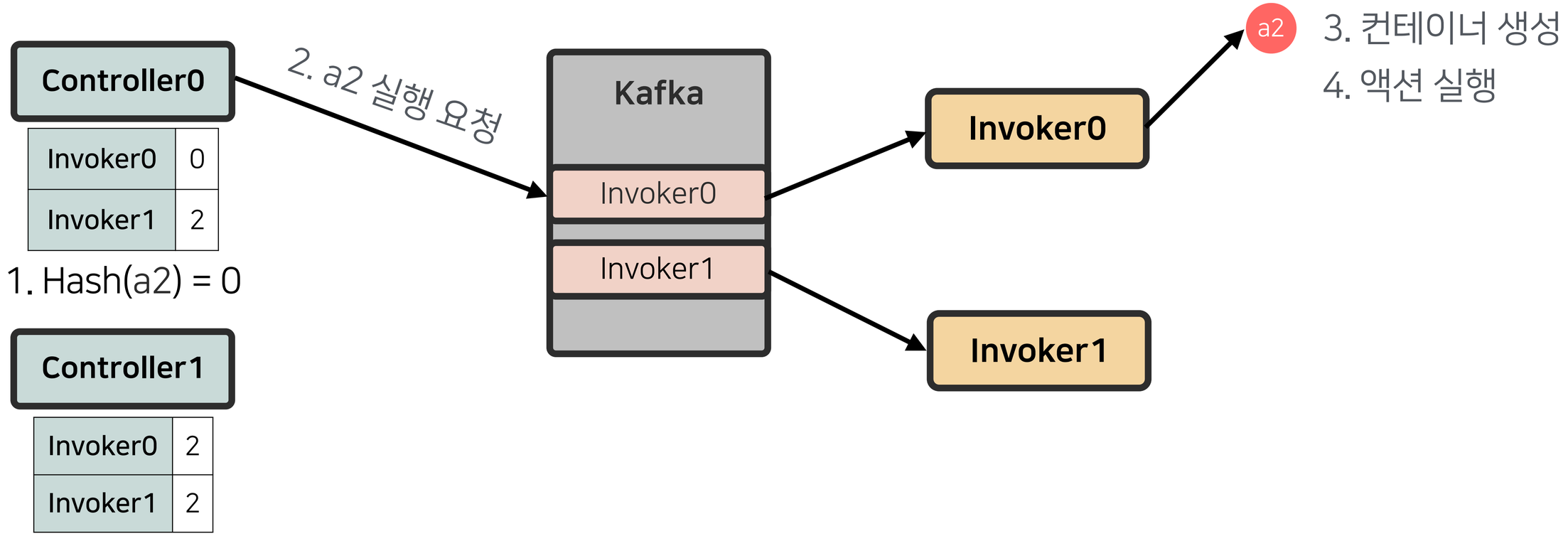


컨테이너의 재사용률이 최대가 됨

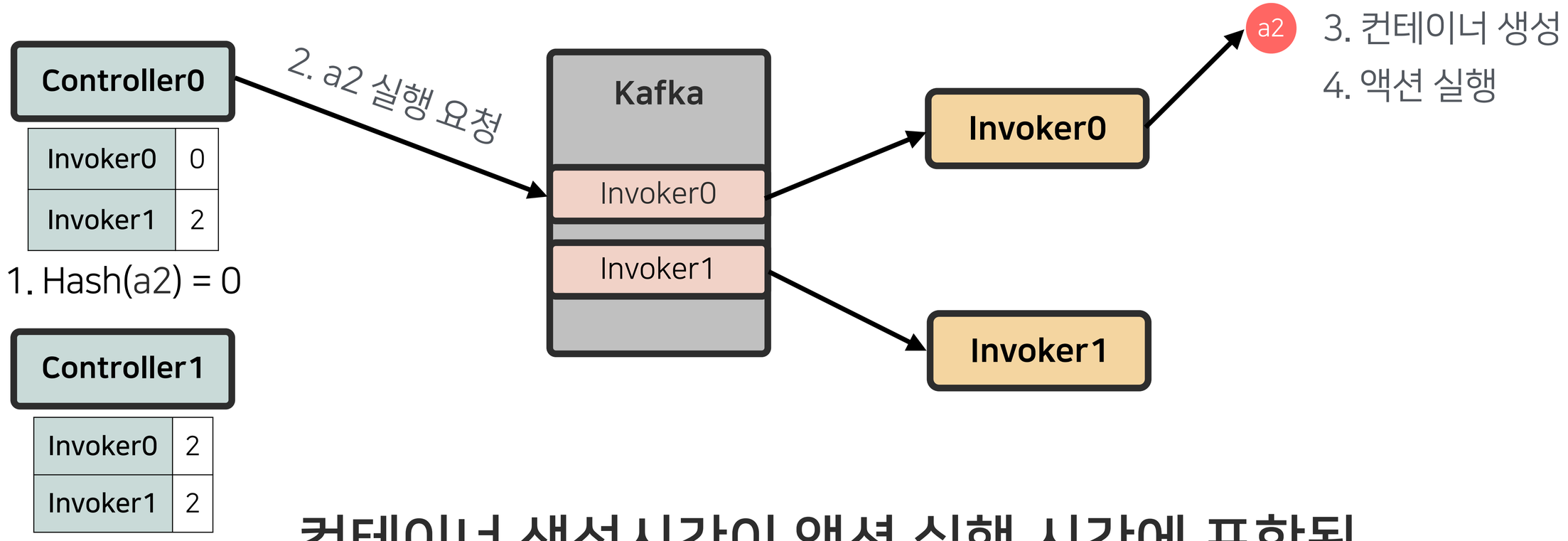
기존 방식 - 컨테이너의 생성이 실행에 영향



기존 방식 - 컨테이너의 생성이 실행에 영향

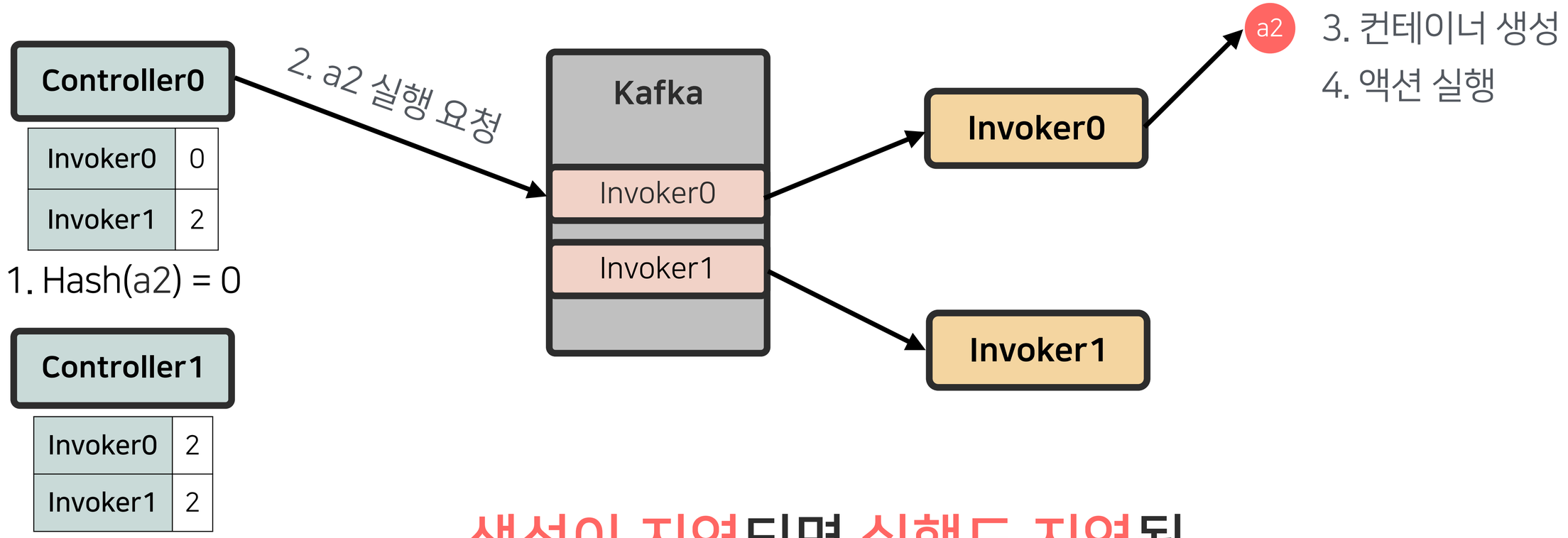


기존 방식 - 컨테이너의 생성이 실행에 영향



컨테이너 생성시간이 액션 실행 시간에 포함됨

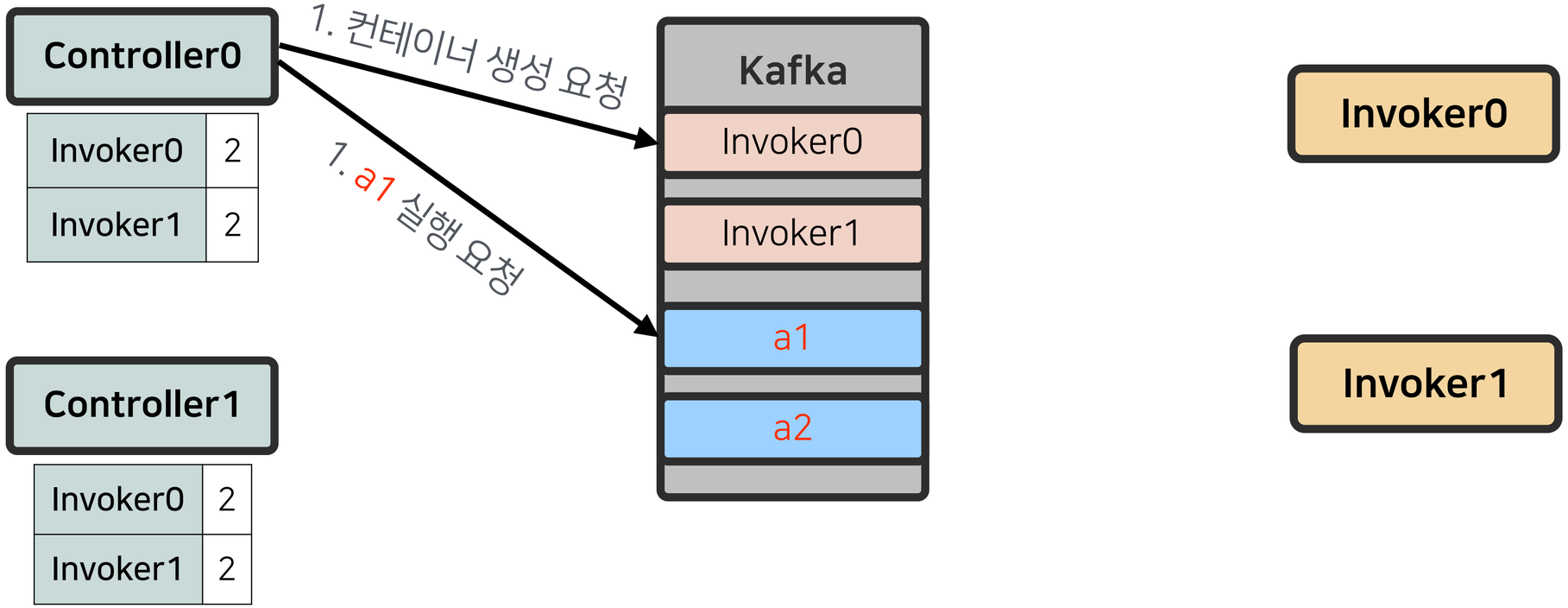
기존 방식 - 컨테이너의 생성이 실행에 영향



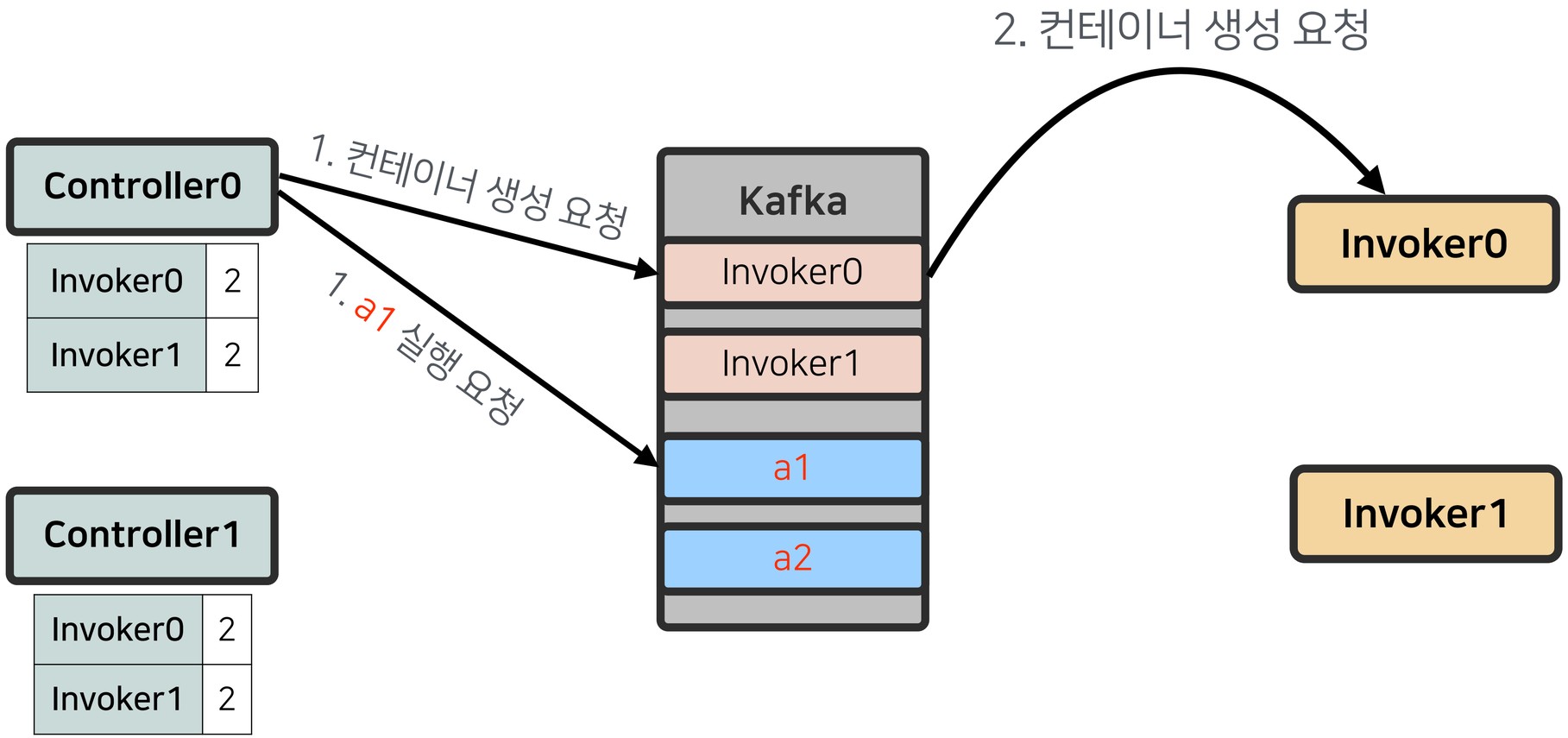
생성이 지연되면 실행도 지연됨

컨테이너의 생성과 액션의 실행을 분리

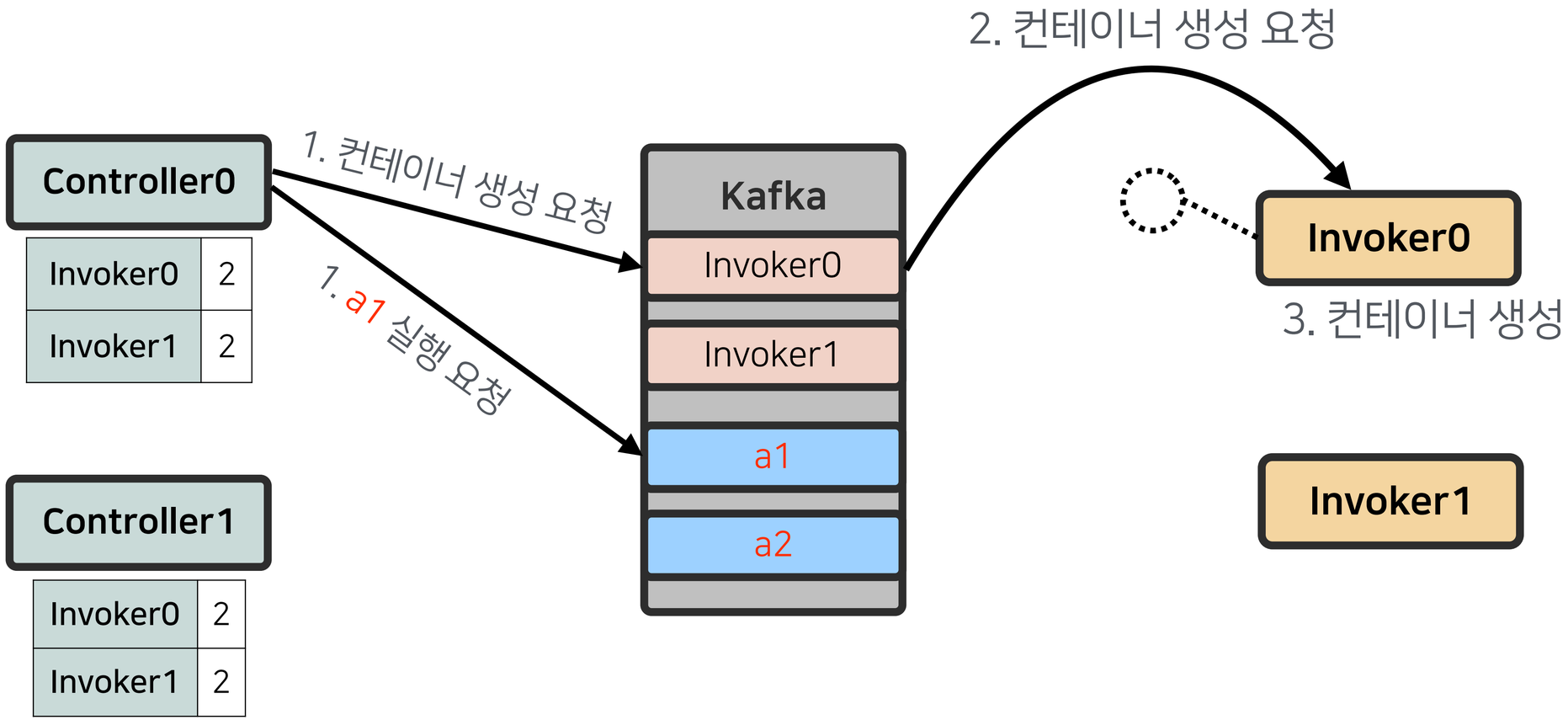
DEVIEW 2019



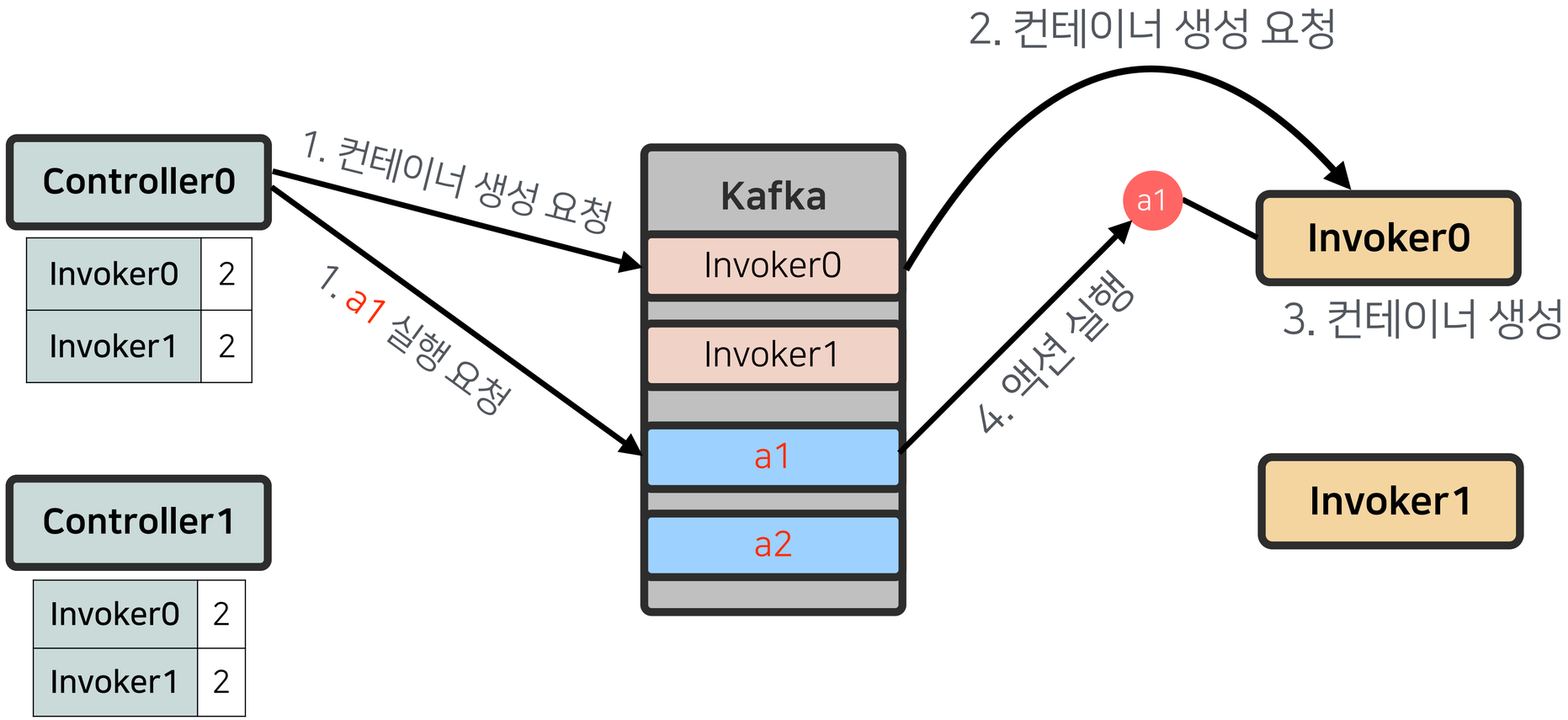
컨테이너의 생성과 액션의 실행을 분리



컨테이너의 생성과 액션의 실행을 분리

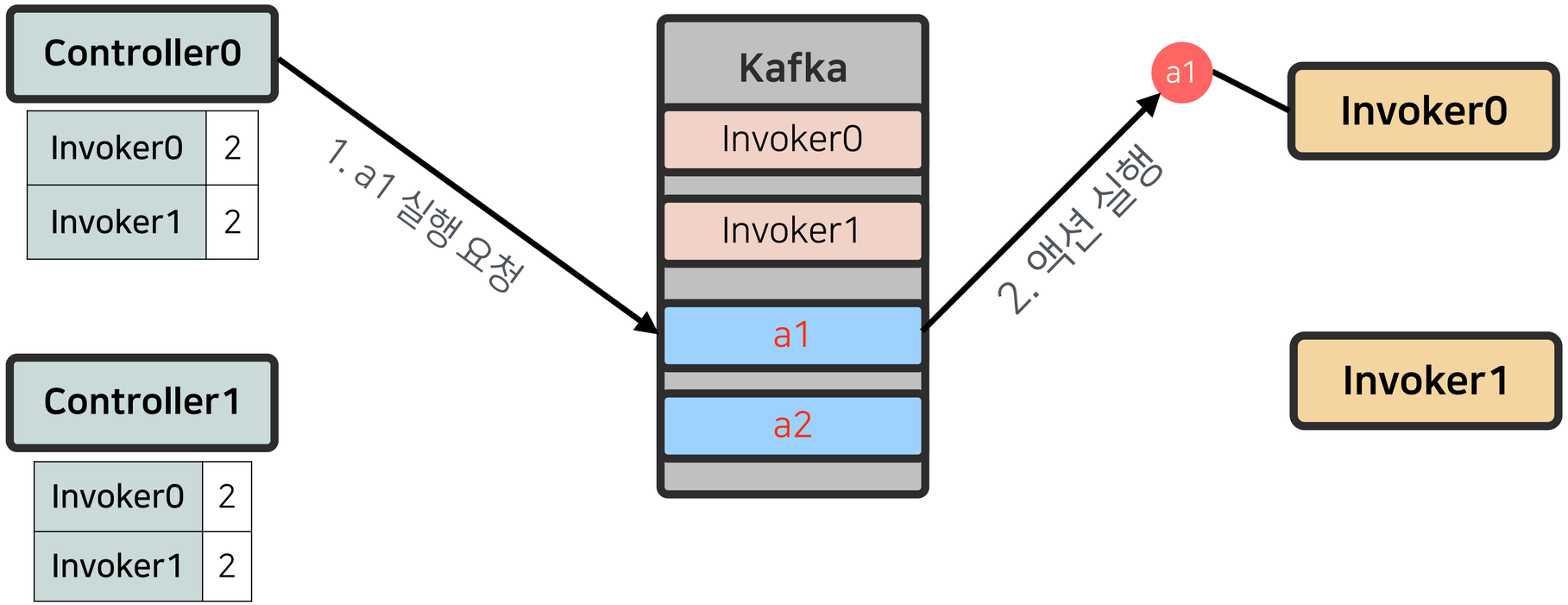


컨테이너의 생성과 액션의 실행을 분리

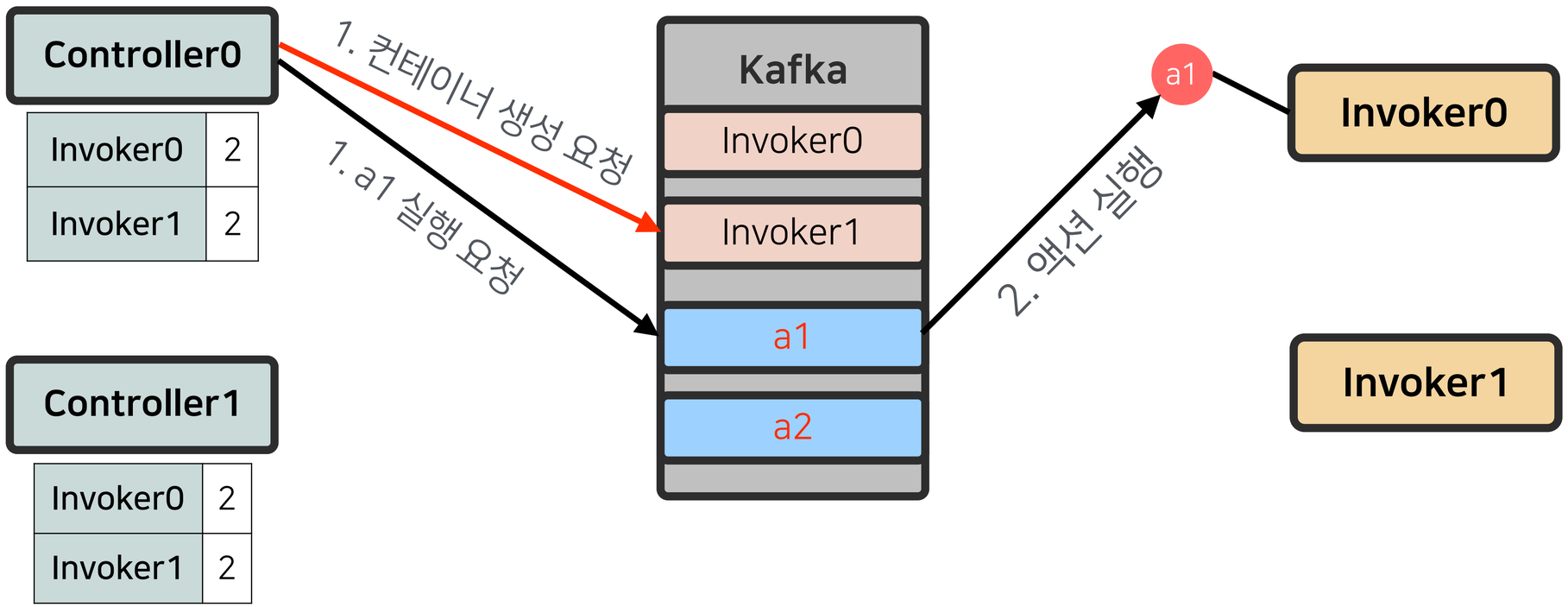


컨테이너의 생성과 액션의 실행을 분리

DEVIEW 2019

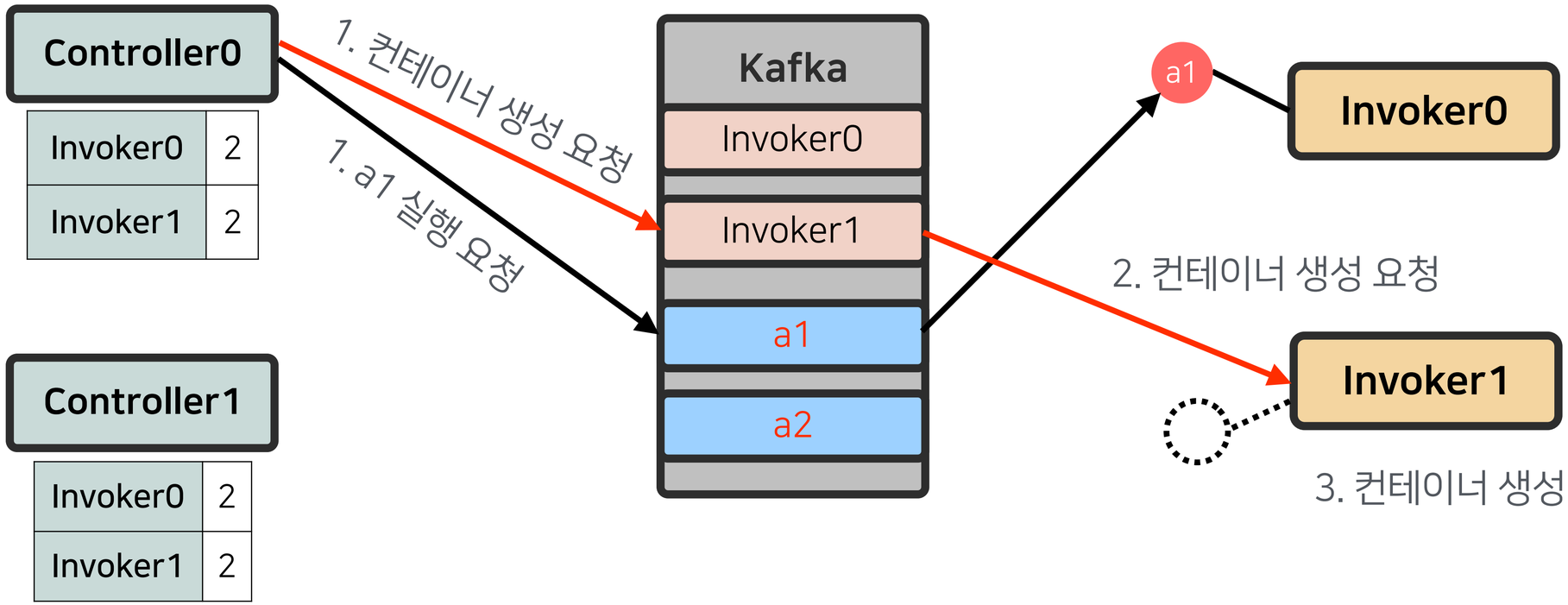


컨테이너의 생성과 액션의 실행을 분리

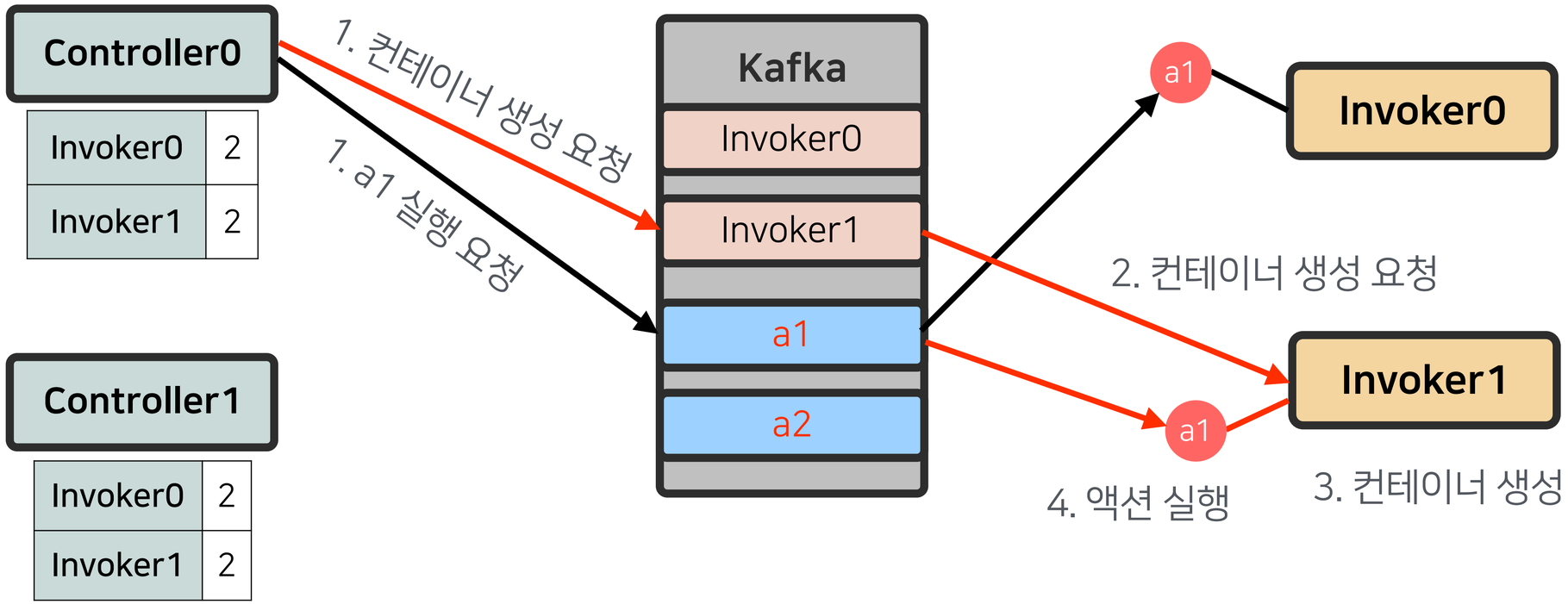


컨테이너의 생성과 액션의 실행을 분리

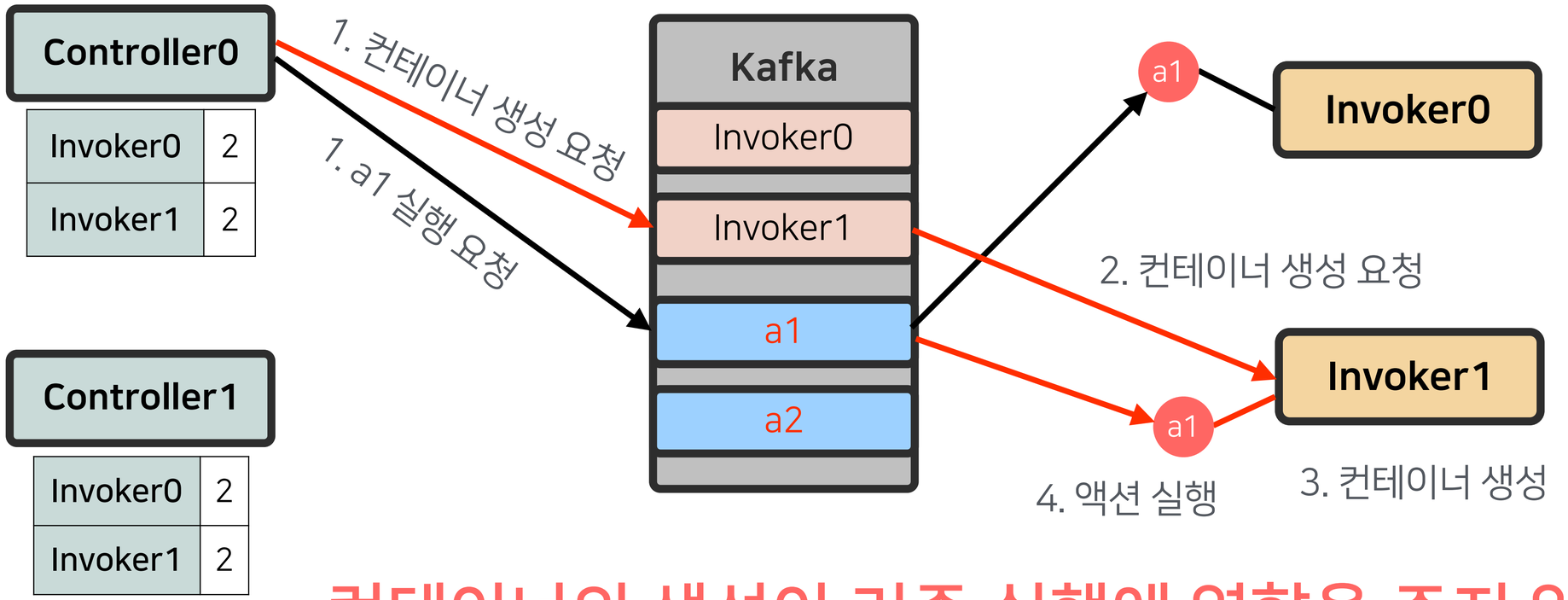
DEVIEW 2019



컨테이너의 생성과 액션의 실행을 분리

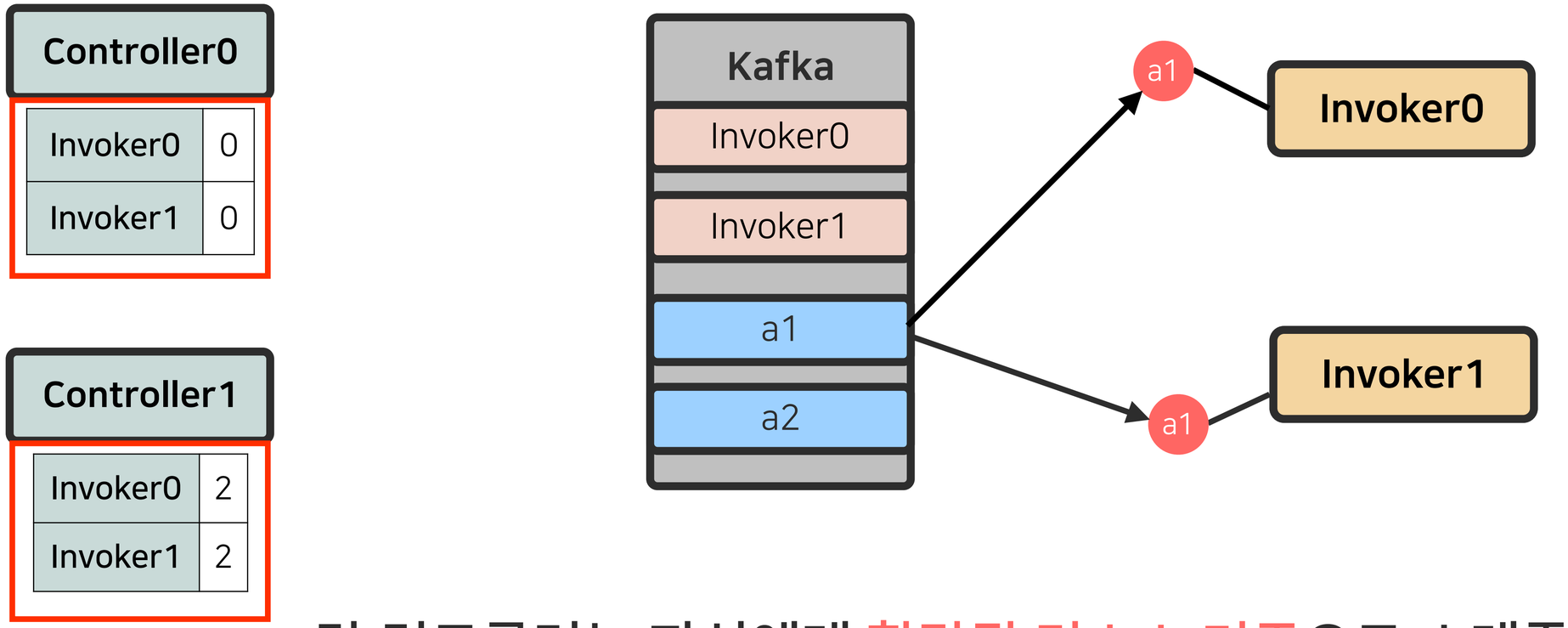


컨테이너의 생성과 액션의 실행을 분리



컨테이너의 생성이 기존 실행에 영향을 주지 않음

신규 컴포넌트 도입 - ETCD



각 컨트롤러는 자신에게 할당된 리소스 기준으로 스케쥴링

신규 컴포넌트 도입 - ETCD

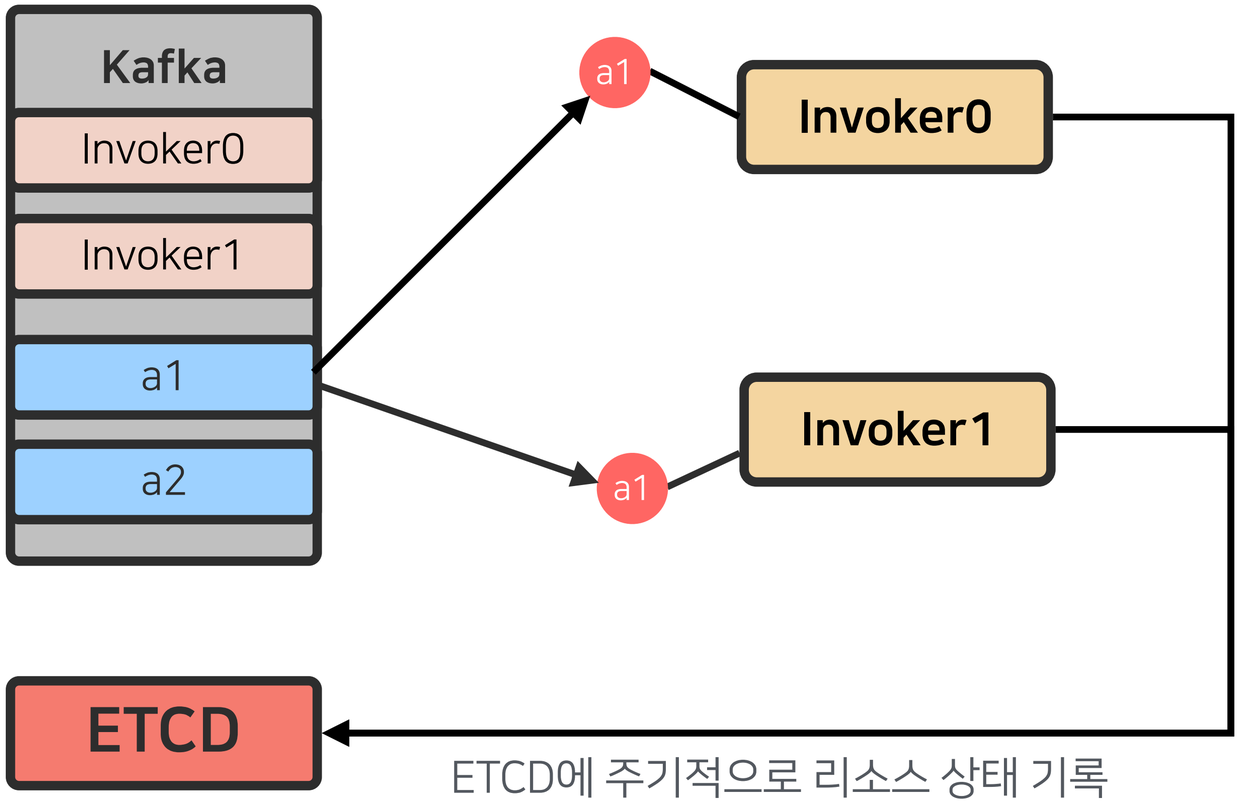
DEVIEW 2019

Controller0

Invoker0	2
Invoker1	2

Controller1

Invoker0	2
Invoker1	2



신규 컴포넌트 도입 - ETCD

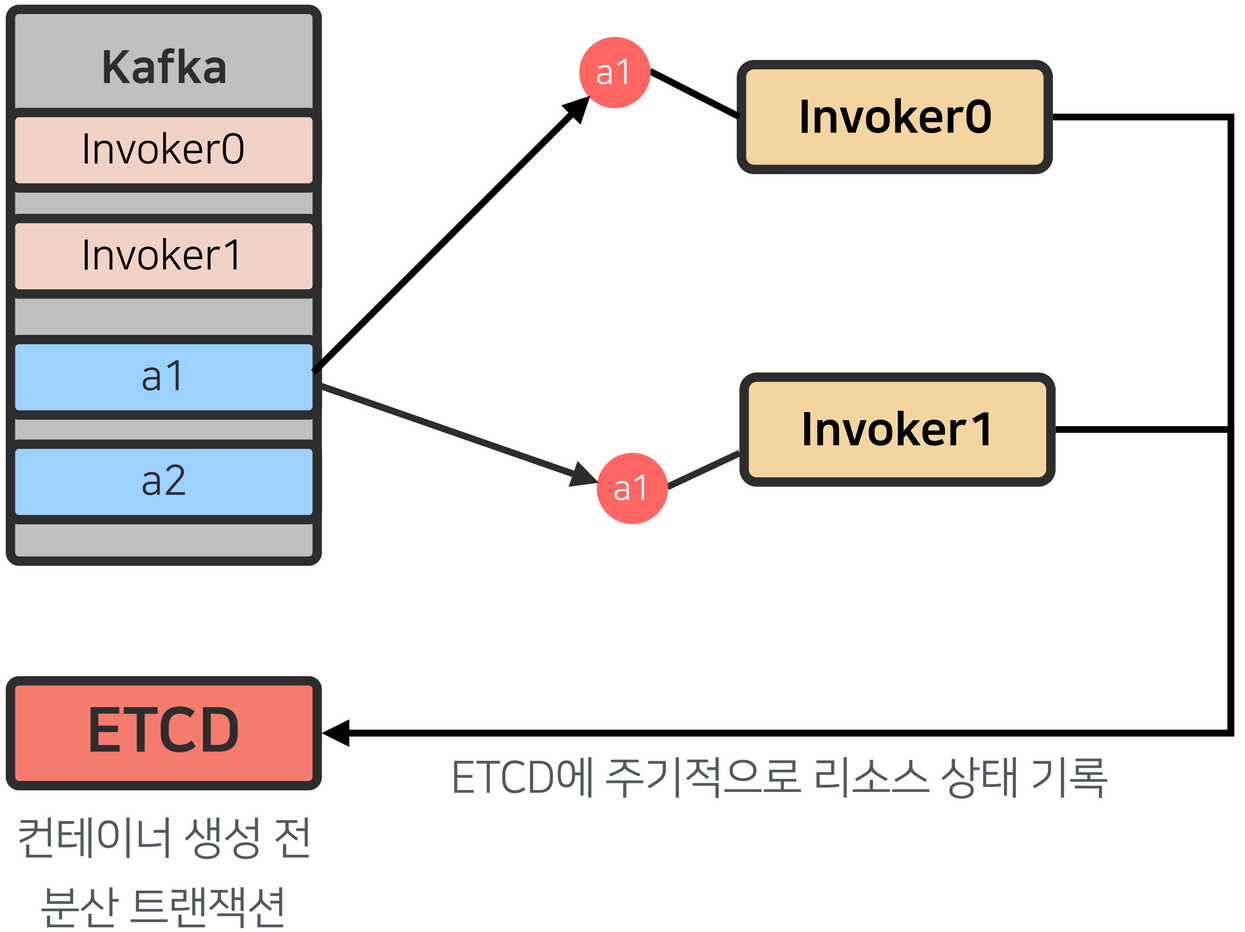
DEVIEW 2019

Controller0

Invoker0	2
Invoker1	2

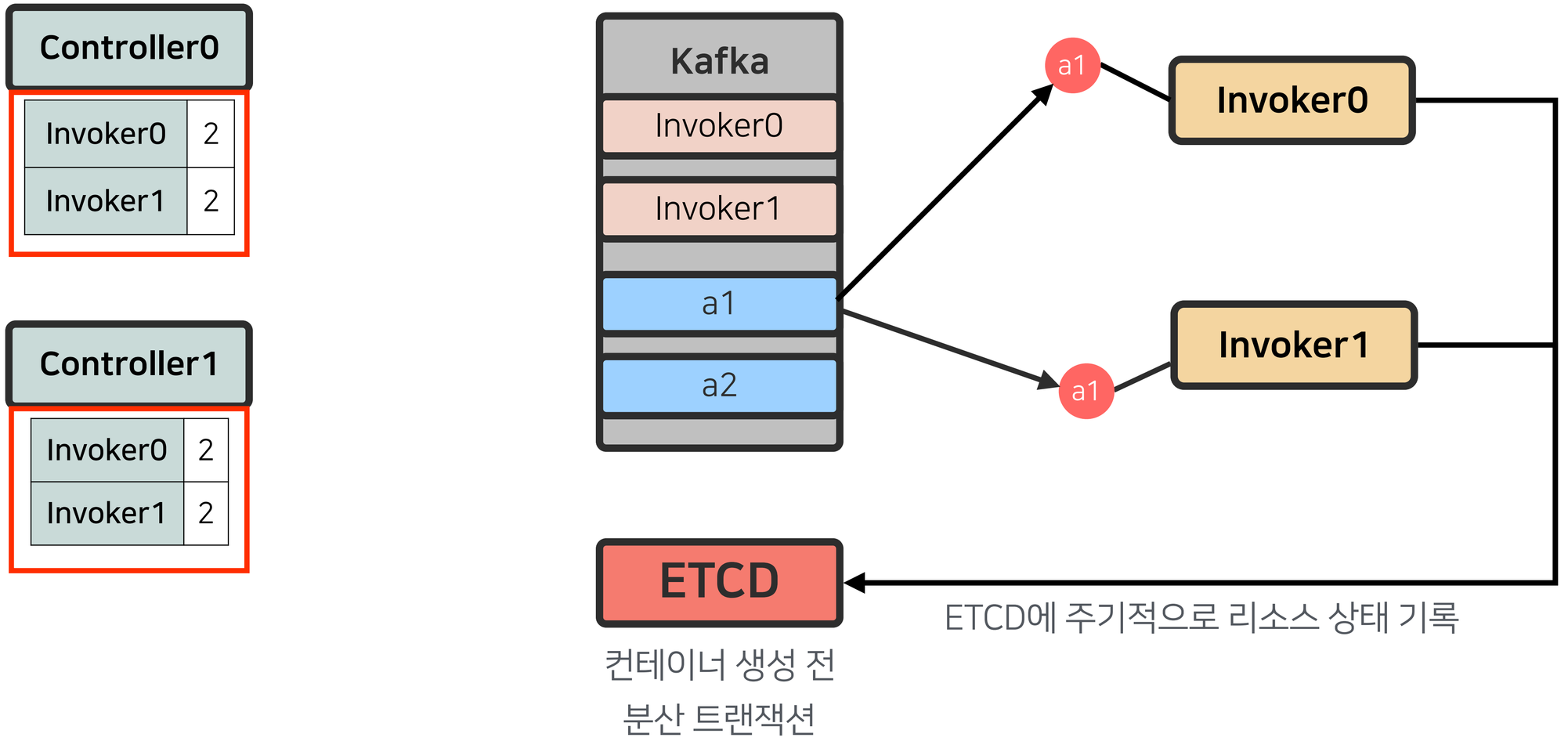
Controller1

Invoker0	2
Invoker1	2

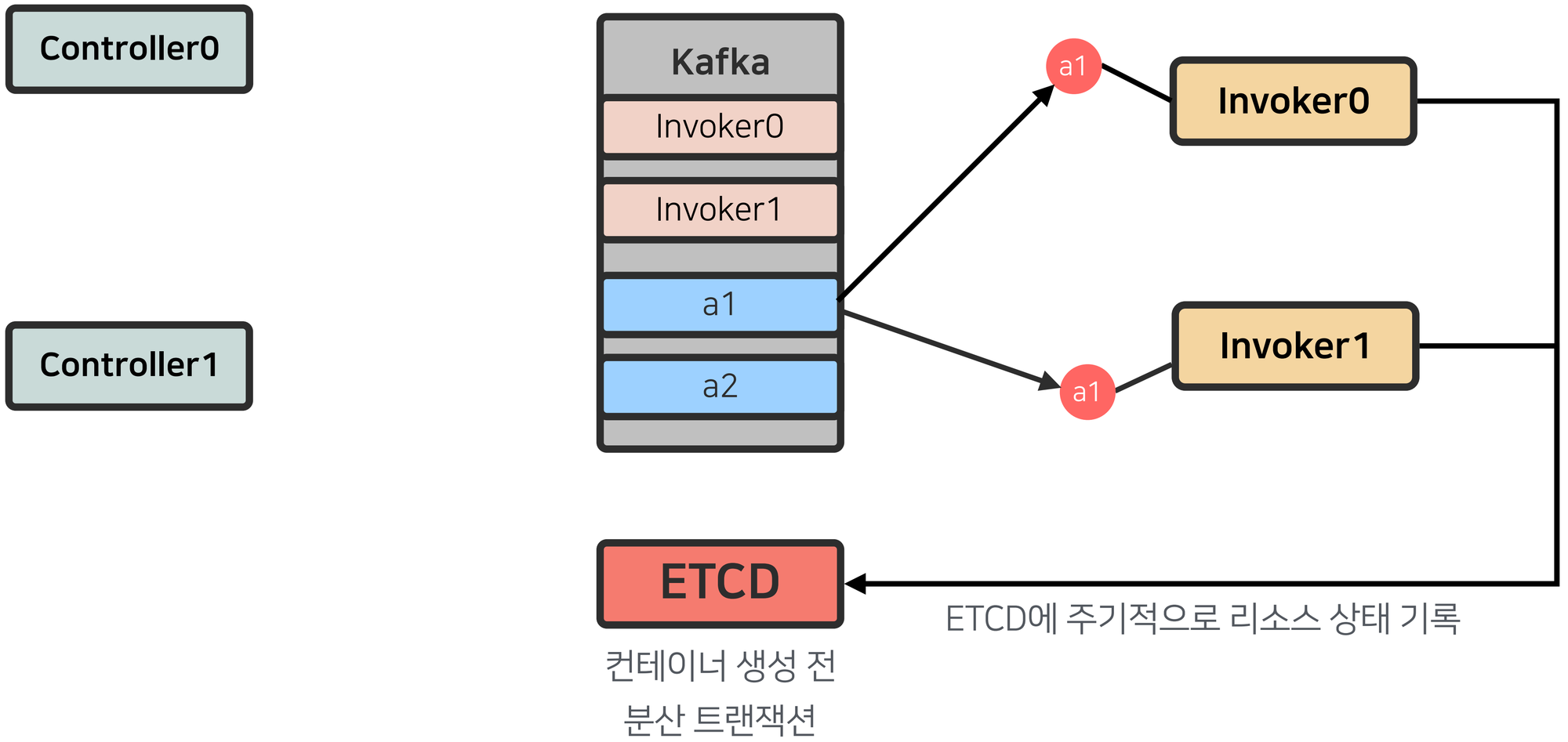


컨테이너 생성 전
분산 트랜잭션

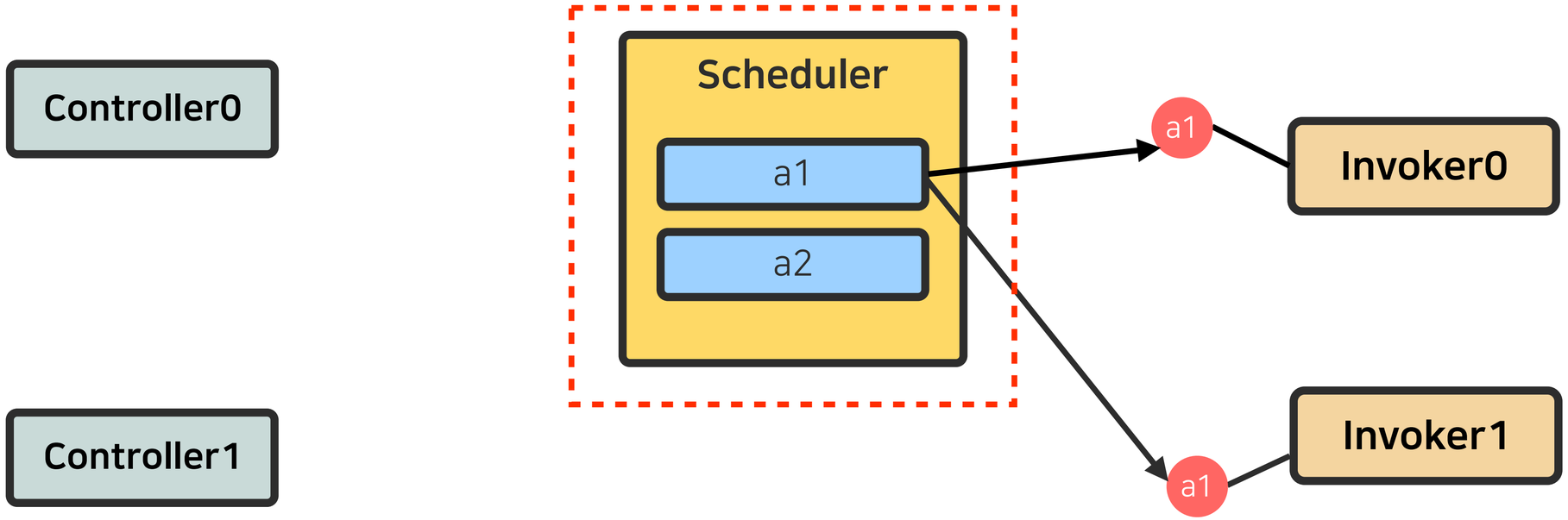
신규 컴포넌트 도입 - ETCD



신규 컴포넌트 도입 - ETCD



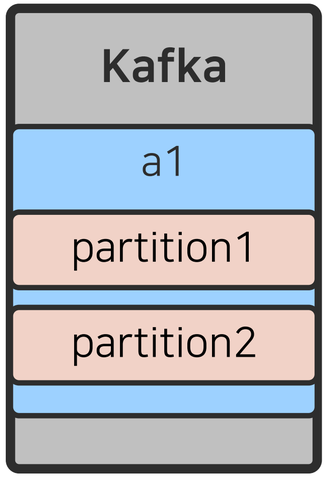
신규 컴포넌트 도입 - Scheduler



Kafka를 대체하는 큐를 직접 구현

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

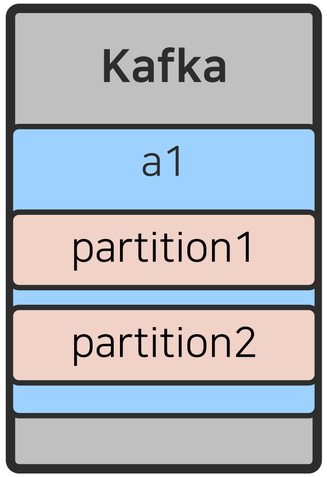
DEVIEW 2019



Kafka 토픽은 파티션으로 이루어짐

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

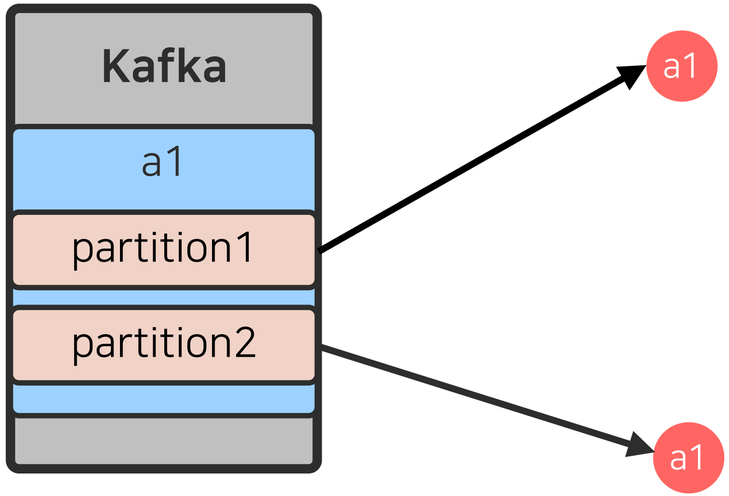
DEVIEW 2019



파티션은 parallelism의 단위

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

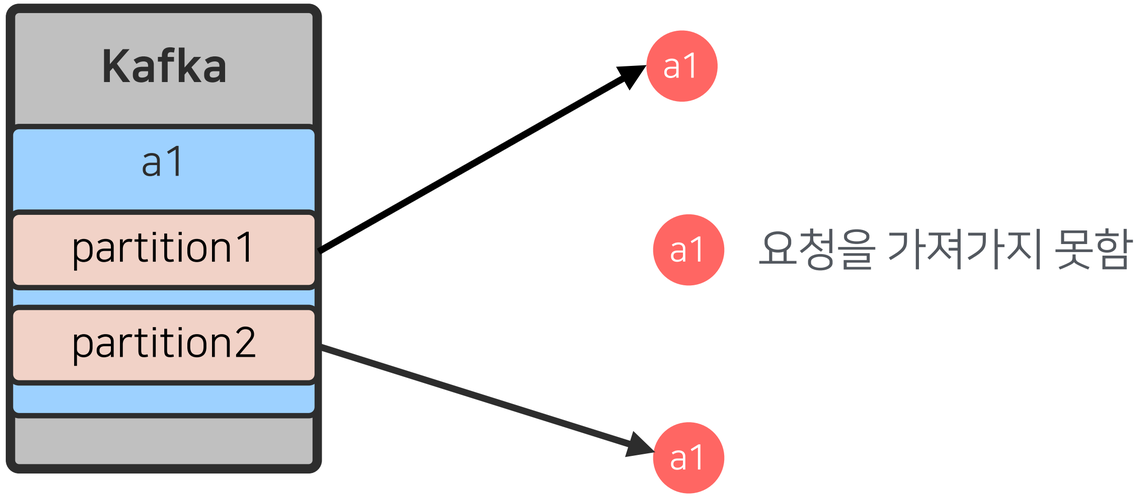
DEVIEW
2019



컨슈머 수와 동일한 수의 파티션 필요

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

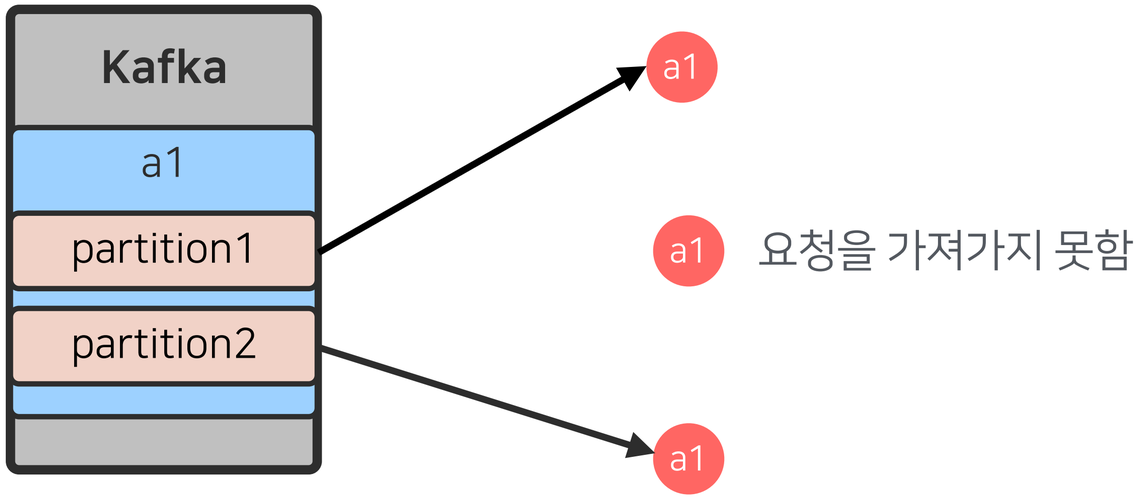
DEVIEW 2019



컨슈머 수와 동일한 수의 파티션 필요

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

DEVIEW 2019

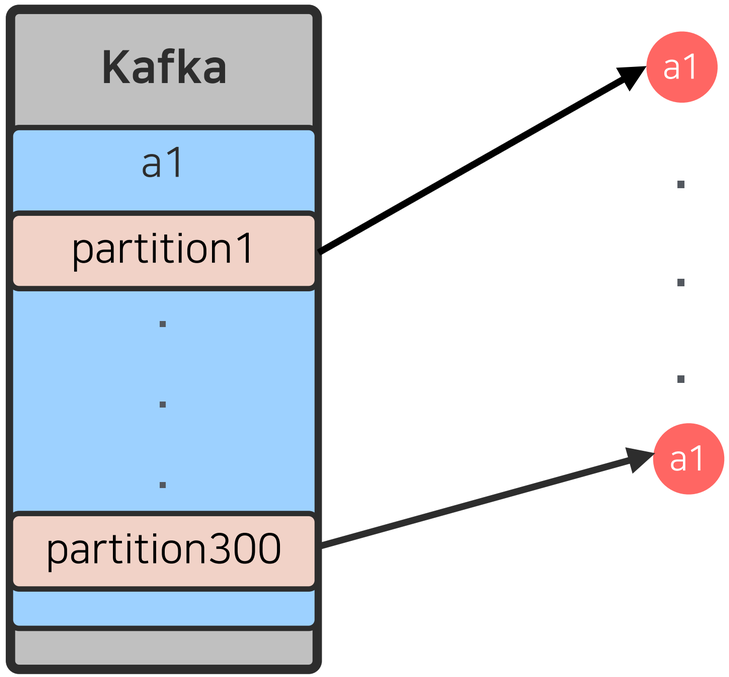


파티션은 동적으로 변경이 어려움

(변경시, 수초간 단절 발생)

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

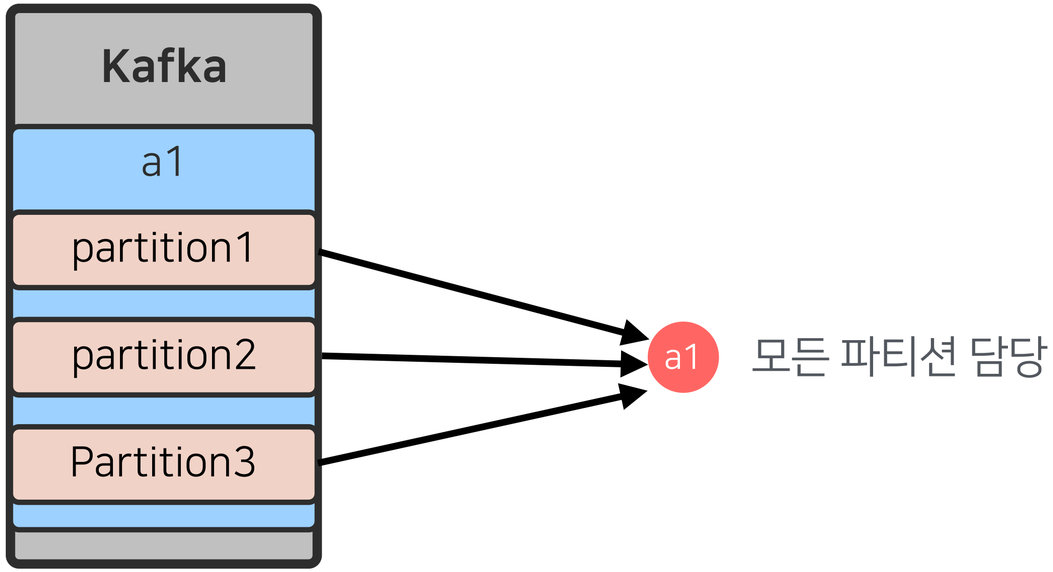
DEVIEW 2019



미리 충분히 크게 늘려 놓아야함

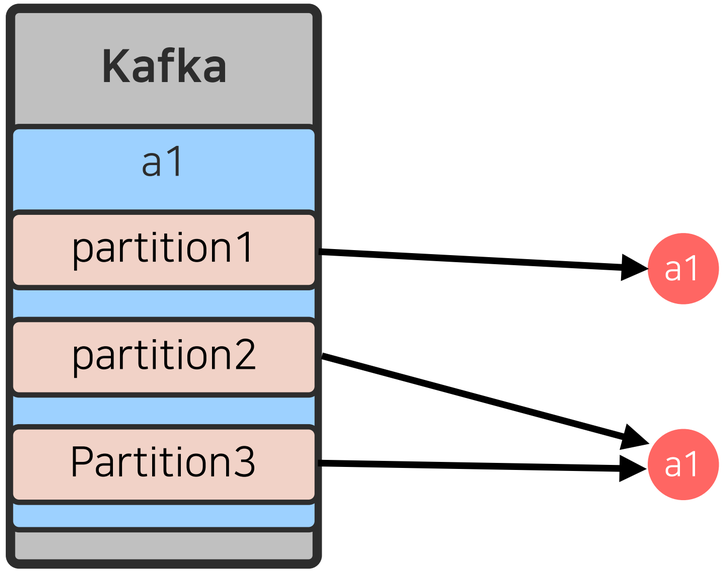
신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

DEVIEW 2019



신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

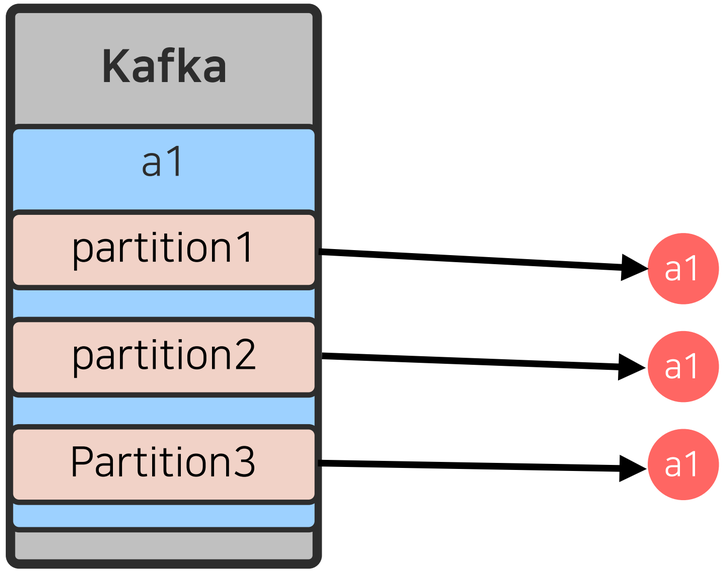
DEVIEW 2019



파티션을 나누어 가짐

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

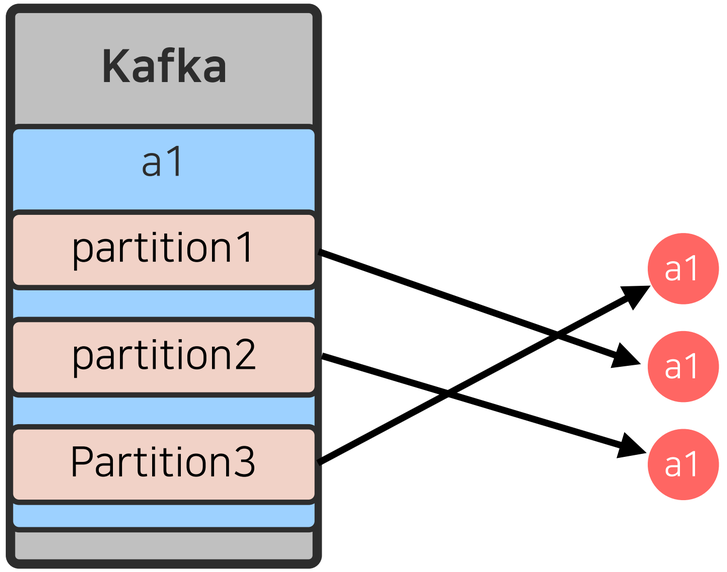
DEVIEW 2019



파티션을 나누어 가짐

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

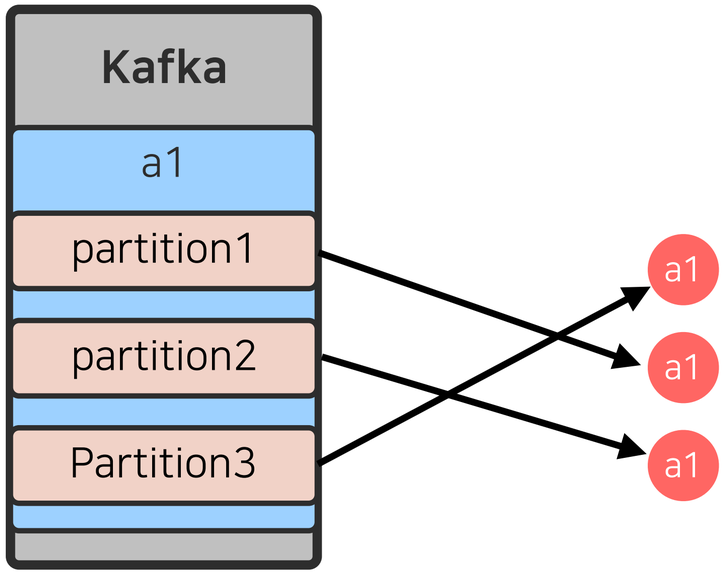
DEVIEW 2019



파티션을 나누어 가짐

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

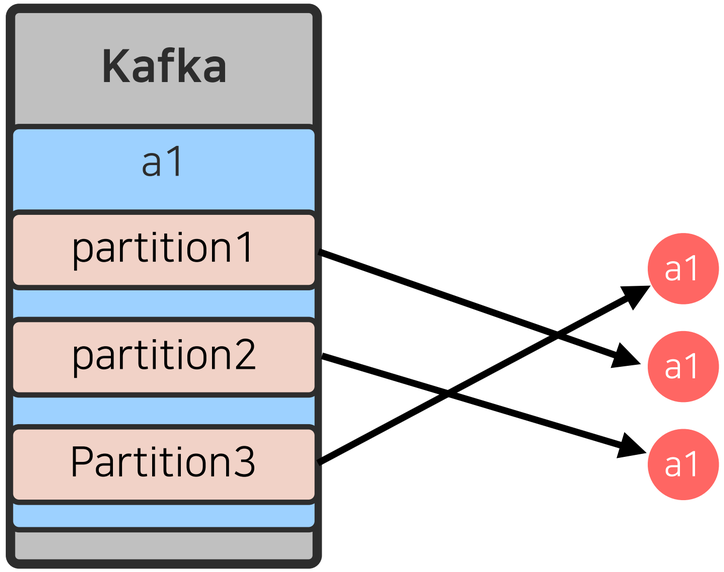
DEVIEW 2019



특정 파티션을 특정 컨슈머에게 할당

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

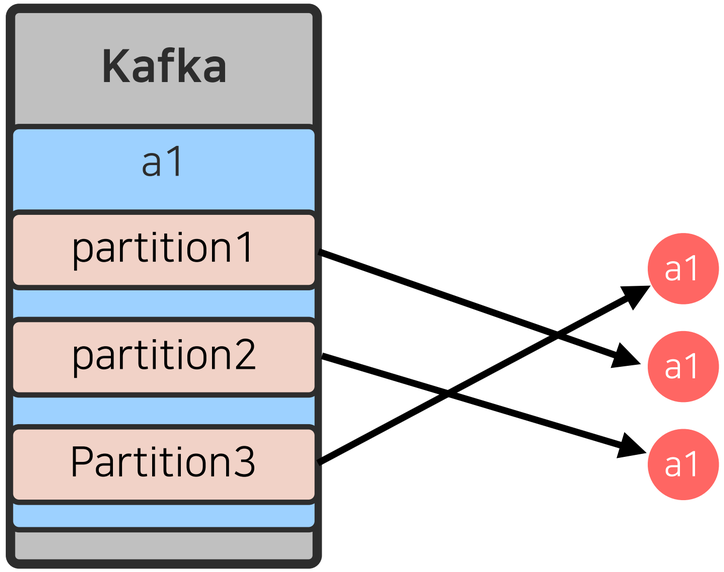
DEVIEW 2019



Consumer Rebalancing

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

DEVIEW 2019

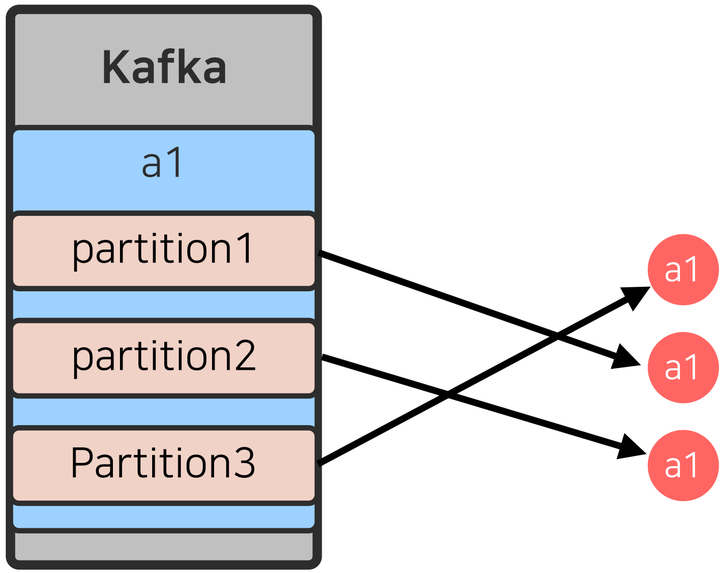


Consumer Rebalancing

Consumer(컨테이너)가 추가될 때마다 발생

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

DEVIEW 2019

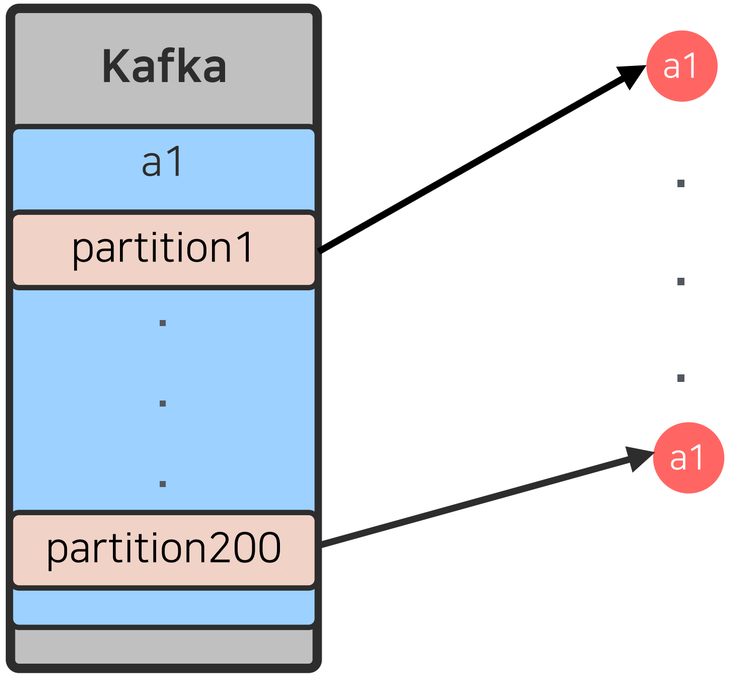


Consumer Rebalancing

컨슈머 개수에 비례한 시간이 걸림

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

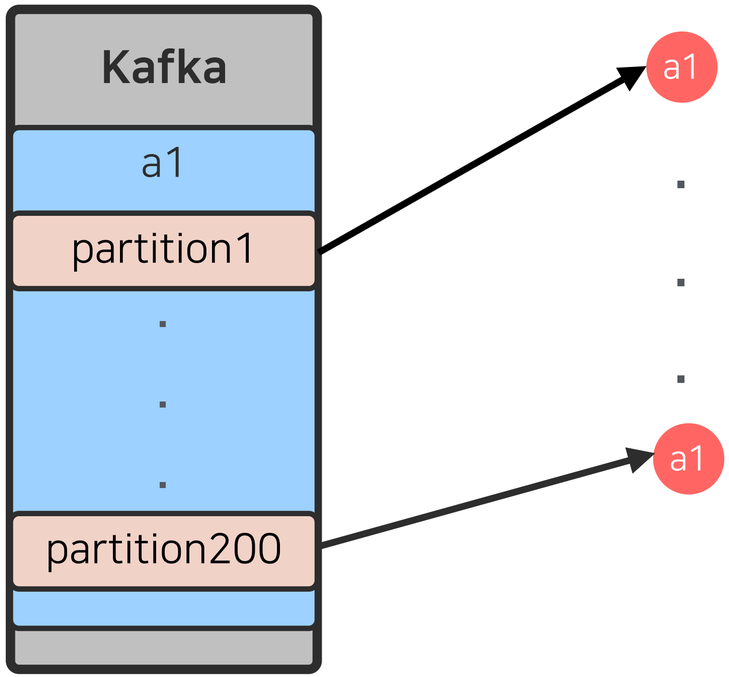
DEVIEW 2019



파티션 200개를 rebalancing하는데 50초 이상 소요

신규 컴포넌트 도입 - Kafka를 쓰지 못한 이유

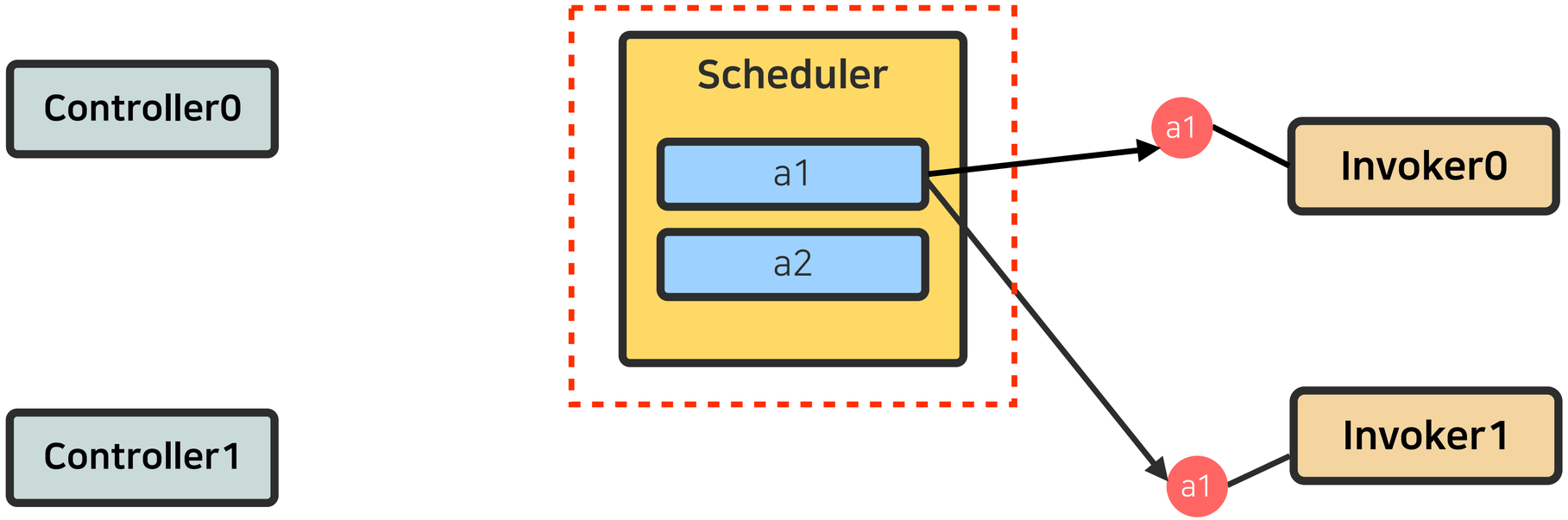
DEVIEW 2019



Routing(Rebalancing)을 우리가 결정할수 없음

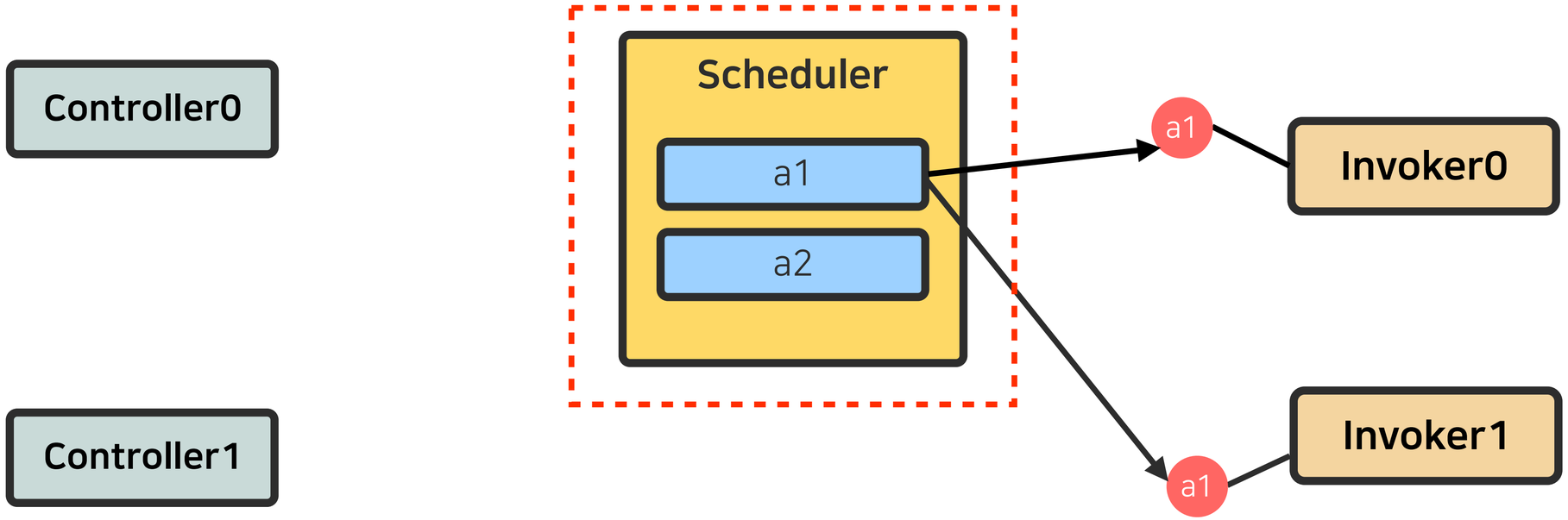
신규 컴포넌트 도입 - Scheduler

DEVIEW
2019



Kafka를 대체하는 큐를 직접 구현

신규 컴포넌트 도입 - Scheduler



실행요청에 대한 메시지 큐 역할

컨테이너의 생성 시점을 결정

Routing에 대한 full control을 가짐

결론

DEVIEW
2019

액션별 메시지 큐 도입

액션간 간섭 문제 해결

결론

액션별 메시지 큐 도입

Pull 기반 메시지 전송

액션간 간섭 문제 해결

컨테이너의 위치를 고려하지 않음

결론

DEVIEW
2019

액션별 메시지 큐 도입

Pull 기반 메시지 전송

액션의 실행과 컨테이너의 생성을 분리

액션간 간섭 문제 해결

컨테이너의 위치를 고려하지 않음

컨테이너 생성이 실행을 지연시키지 않음

결론

액션별 메시지 큐 도입

Pull 기반 메시지 전송

액션의 실행과 컨테이너의 생성을 분리

ETCD 기반으로 리소스 공유 및 분산 트랜잭션

액션간 간섭 문제 해결

컨테이너의 위치를 고려하지 않음

컨테이너 생성이 실행을 지연시키지 않음

리소스를 글로벌하게 관리

결론

액션별 메시지 큐 도입

Pull 기반 메시지 전송

액션의 실행과 컨테이너의 생성을 분리

ETCD 기반으로 리소스 공유 및 분산 트랜잭션

스케쥴러 컴포넌트 구현

액션간 간섭 문제 해결

컨테이너의 위치를 고려하지 않음

컨테이너 생성이 실행을 지연시키지 않음

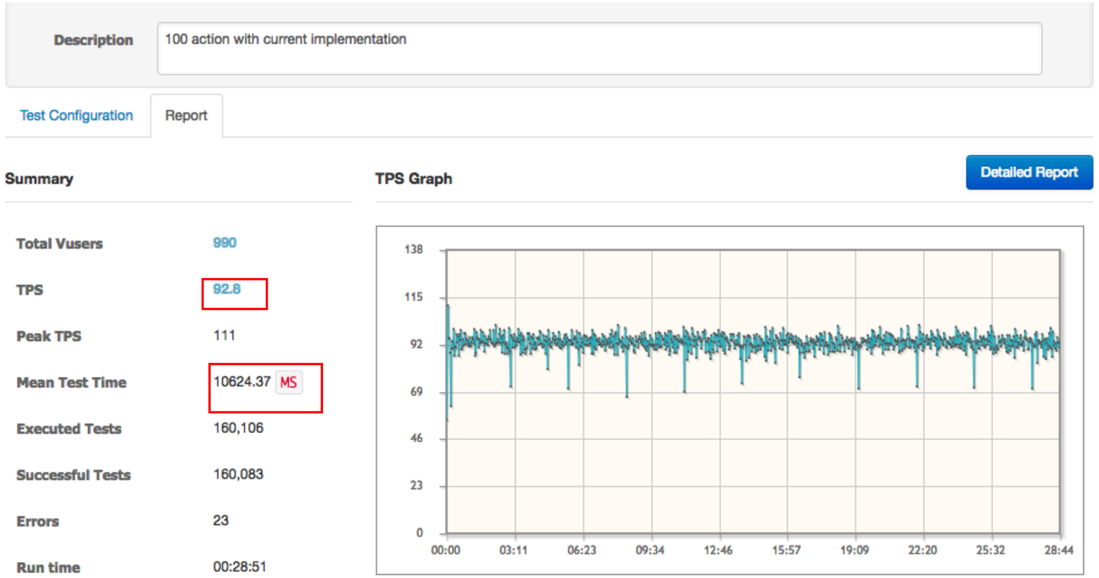
리소스를 글로벌하게 관리

Routing에 대한 Full Control을 가짐

성능 비교 벤치마크

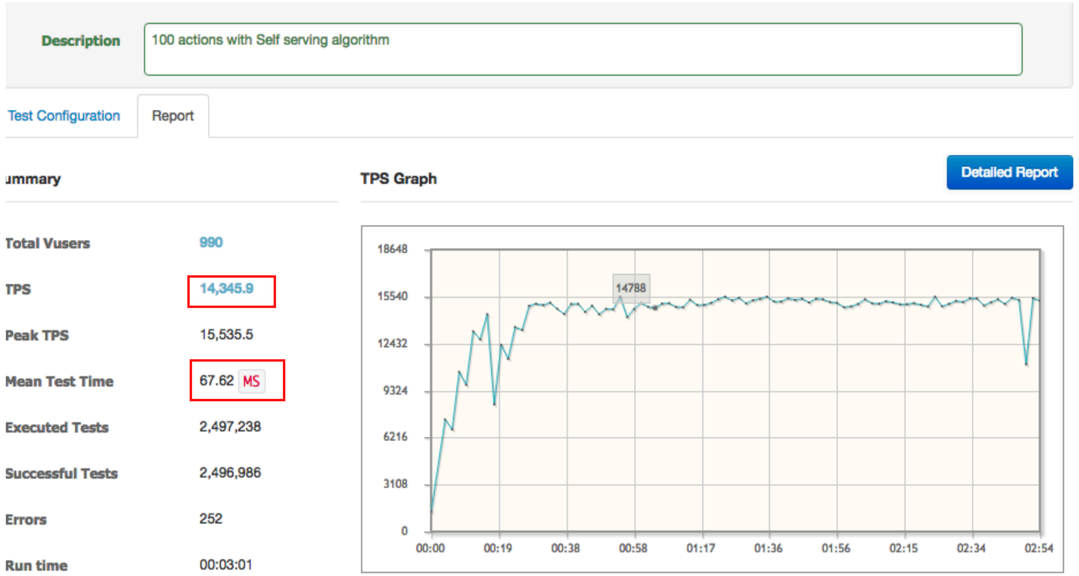
DEVIEW 2019

오픈소스



90 TPS

신규 스케쥴러

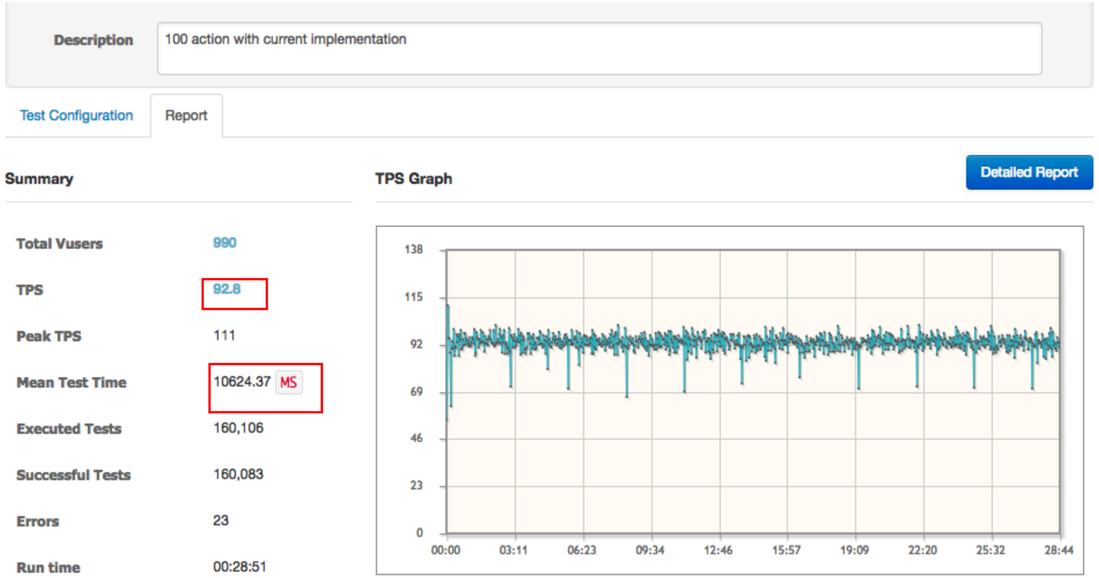


14,000 TPS

성능 비교 벤치마크

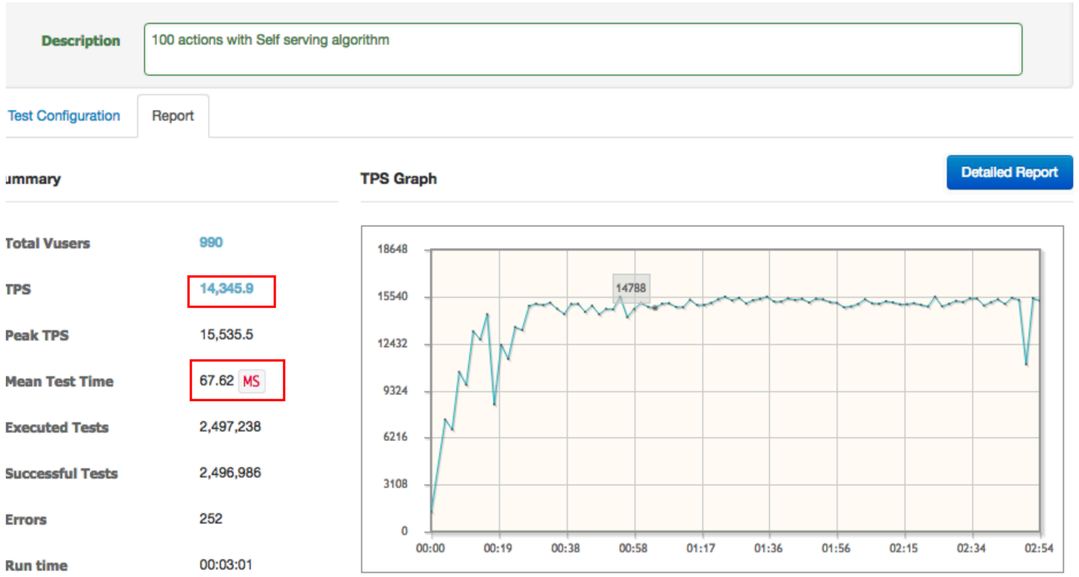
DEVIEW 2019

오픈소스



90 TPS

신규 스케쥴러



14,000 TPS

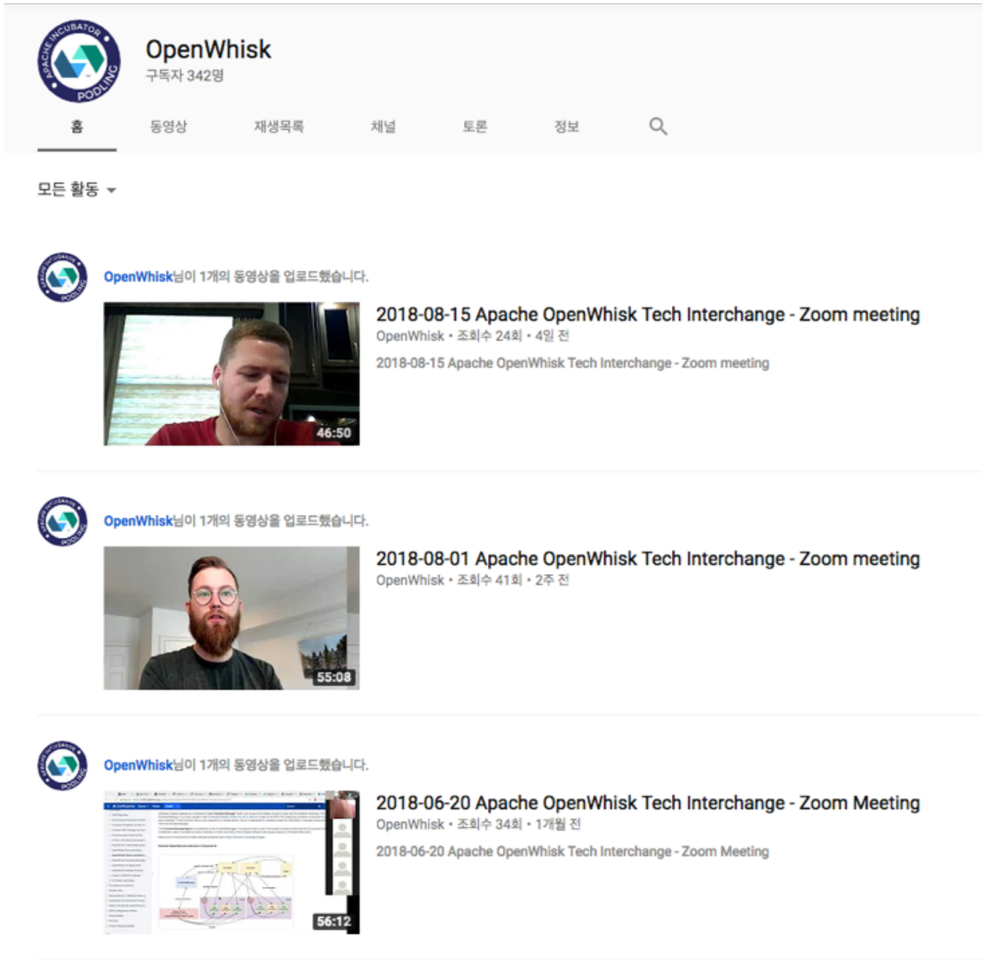
155배

DEVIEW
2019

오픈소스 컨트리뷰션

Apache OpenWhisk - Youtube 채널

DEVIEW 2019



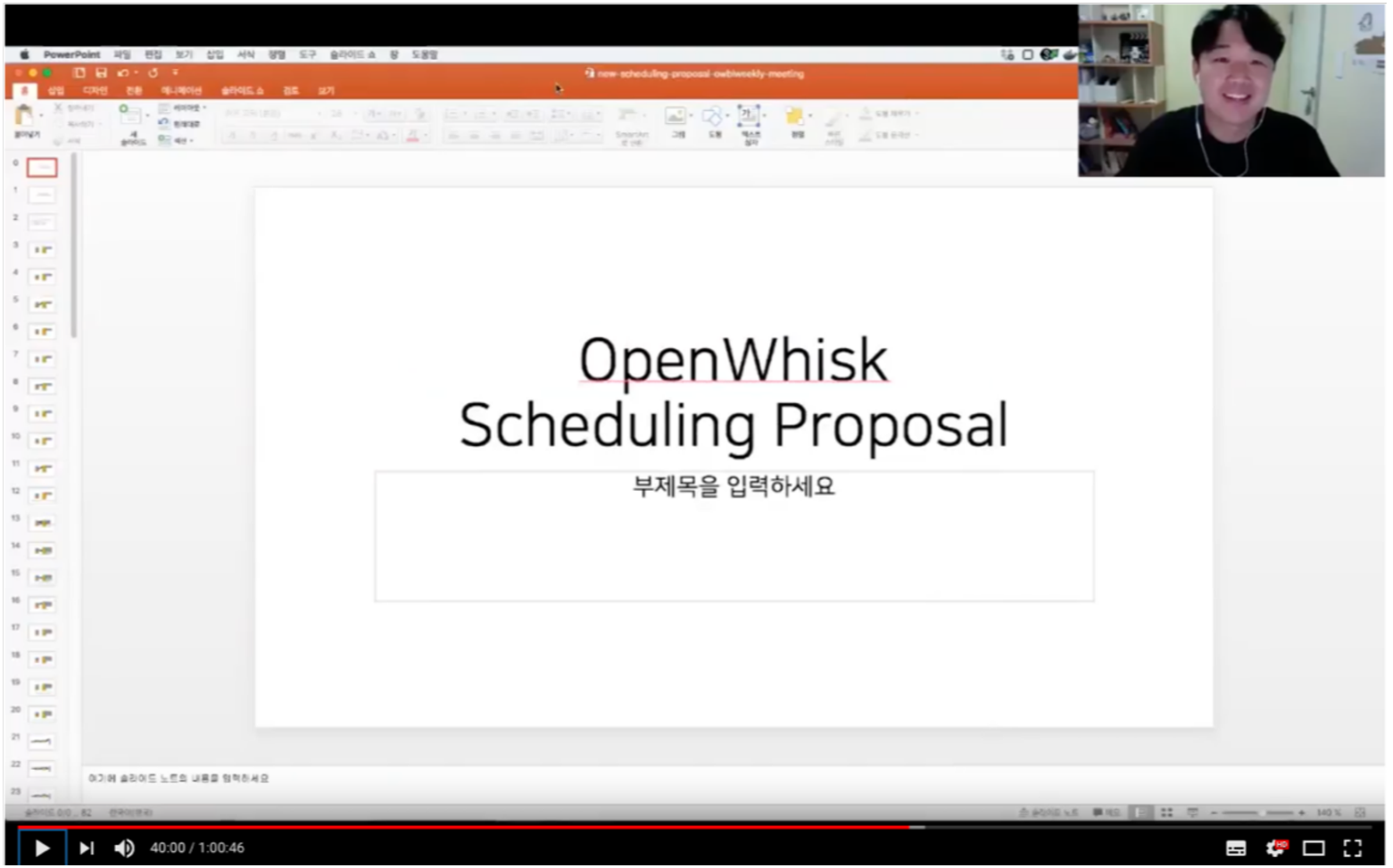
Tech Interchange Call

- 전세계 커뮤니티 멤버들의 화상회의
- 매 2주마다 진행
- 이슈나 신규 기술 등을 공유

https://www.youtube.com/watch?v=cgictUeK-Vk&feature=youtu.be

신규 스케쥴링 proposal 공유

DEVIEW
2019

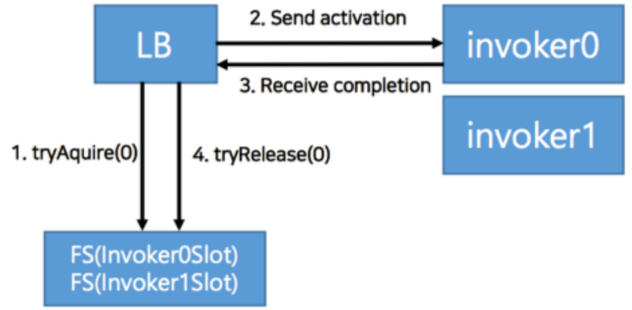


신규 스케쥴링 proposal 공유

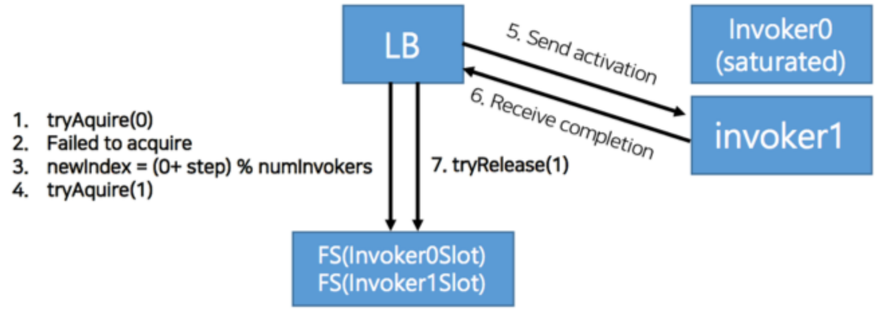
Current Implementation (3/6) – *Forcable Semaphore*

- Loadbalancer chooses invoker based on *ForcableSemaphore*

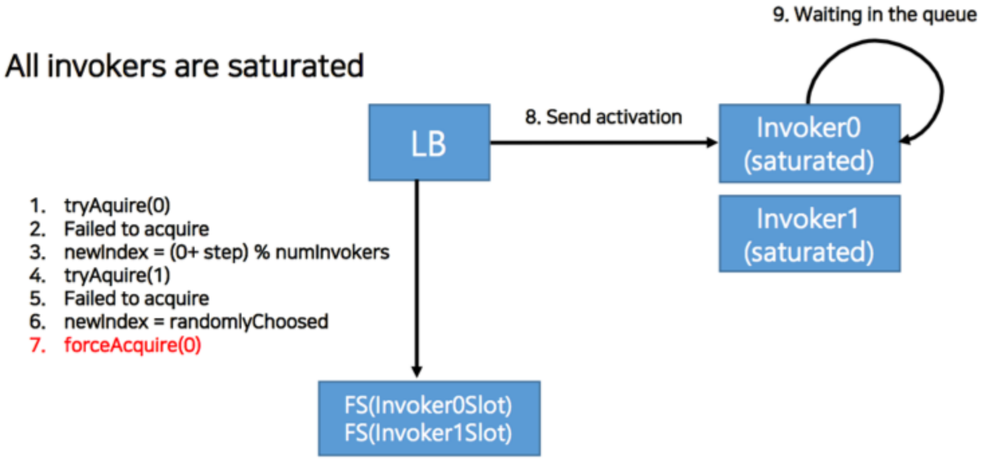
Normal situation



Home Invoker is saturated

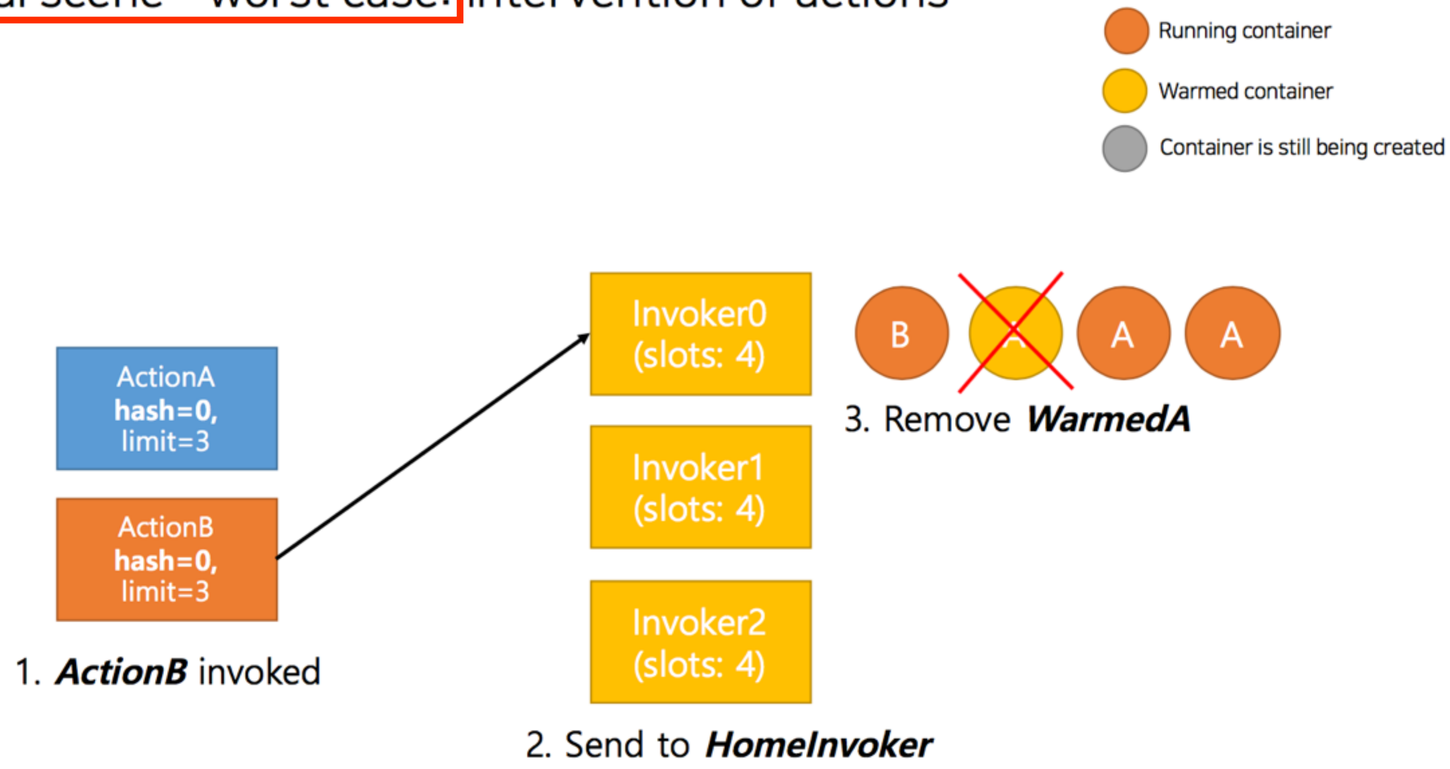


All invokers are saturated



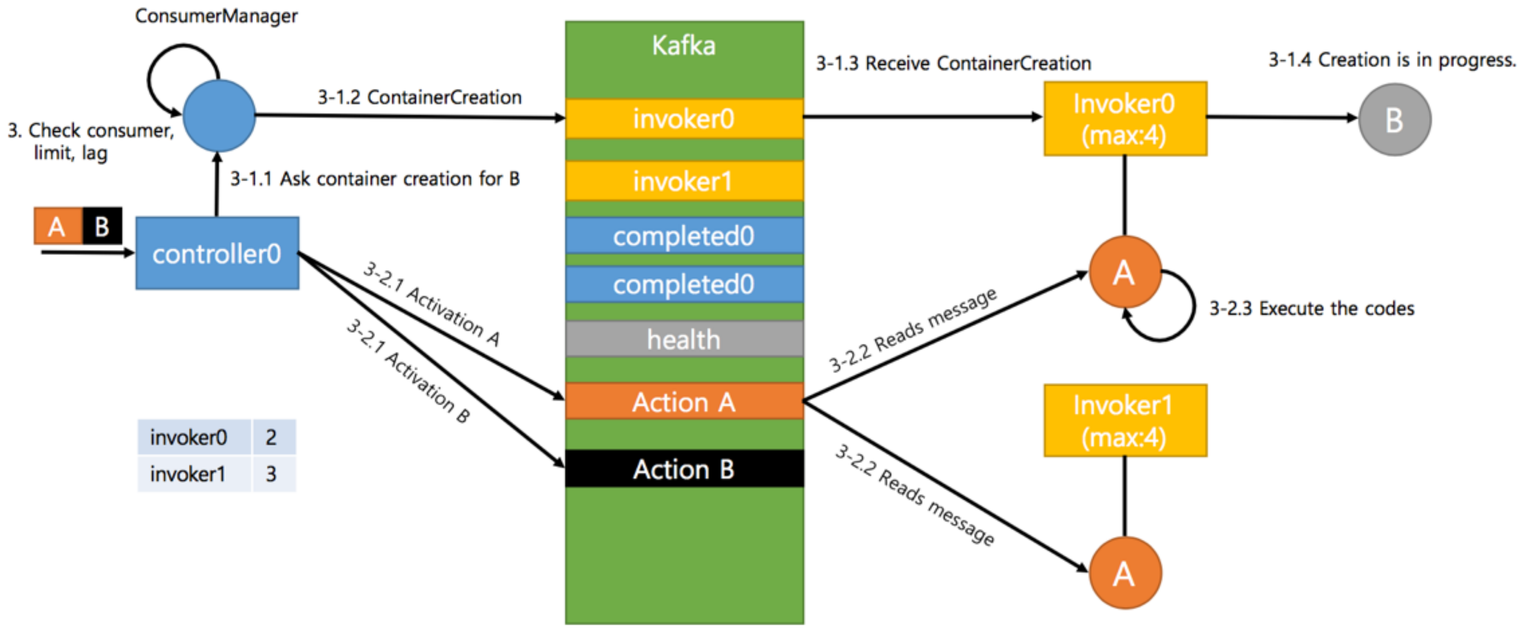
신규 스케쥴링 proposal 공유

Problems in real scene – worst case: intervention of actions



신규 스케쥴링 proposal 공유

Autonomous Container Scheduling Performance Isolation



Container creation does not affect the existing container performance

신규 스케쥴링 proposal 공유

Performance comparison - 100 actions test

Current implementation

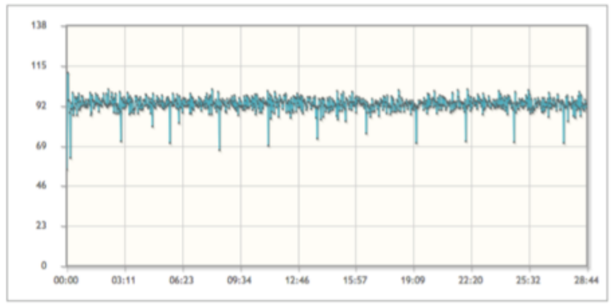
Description: 100 action with current implementation

Test Configuration | Report

Summary

Total Vusers	990
TPS	92.8
Peak TPS	111
Mean Test Time	10624.37 MS
Executed Tests	160,106
Successful Tests	160,083
Errors	23
Run time	00:28:51

TPS Graph — Detailed Report



100 actions with 180 containers

New implementation

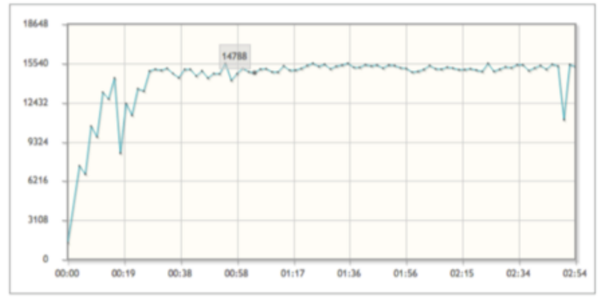
Description: 100 actions with Self serving algorithm

Test Configuration | Report

ummary

Total Vusers	990
TPS	14,345.9
Peak TPS	15,535.5
Mean Test Time	67.62 MS
Executed Tests	2,497,238
Successful Tests	2,496,986
Errors	252
Run time	00:03:01

TPS Graph — Detailed Report



100 actions with 1 containers each(100 containers)

163 times more TPS
158 times faster execution

신규 스케쥴링 proposal 공유

DEVIEW 2019

Confluence 공간

OpenWhisk

- 페이지
- 블로그

공간 바로가기

- Project links

페이지 트리

- Subproject listing
- Policies
- Proposals
  - [WIP] Bug Day
  - [WIP] OpenWhisk on Knative
  - AI Actions
  - **Autonomous Container Sch...**
  - Clustered Singleton Invoker f...
  - Forward Wiki Changes to Co...
  - Illustrating api-experimental
  - Invoker Activation Queueing ...
  - OpenWhisk / Kubernetes pro...
  - OpenWhisk Documentation P...
  - OpenWhisk future architecture
  - OpenWhisk Graduate Manage...
  - OpenWhisk on Kubernetes
  - OpenWhisk Release Process
  - OpenWhisk Website Redesign
  - Project CREDITS proposal
  - Proposal to Include Karate(B...
  - Slack Chat Bot for OpenWhis...

페이지 / OpenWhisk Project Wiki / Proposals

Autonomous Container Scheduling

작성자 : Dominic Kim - 5월 25, 2018

I described the details in the attached file. I will give a speech on this at biweekly meeting on 6th June, 2018.

Main agendas are as follow:

Index

1. Current implementation details
2. Potential issues with current implementation
3. New scheduling algorithm proposal: Autonomous Container Scheduling
4. Review previous issues with new scheduling algorithm
5. Pros and cons of new scheduling algorithm
6. Performance evaluation with prototyping

https://cwiki.apache.org/confluence/display/OPENWHISK/Autonomous+Container+Scheduling

신규 스케쥴링 proposal 공유

DEVIEW 2019

Autonomous Container Scheduler v2

작성자 : Dominic Kim - 11월 28, 2018

This document describes the next version of Autonomous Container Scheduling.

Basics

Currently, discussion and the implementation for future architecture of OW are in progress.

I agree with many parts of the future architecture and that would be the right direction to step up, IMHO.

(I hope I can also participate in the implementation of them.)

It seems, however it will take some time to take those into OW and stabilize them.

What if we can improve the performance of OpenWhisk more than 150 times with the relatively small amount of efforts by adopting new scheduler based on current architecture?

I think it would be worth to take it in until the new implementation is ready.

With SPI abstraction, the new scheduler can even coexist with the existing one.

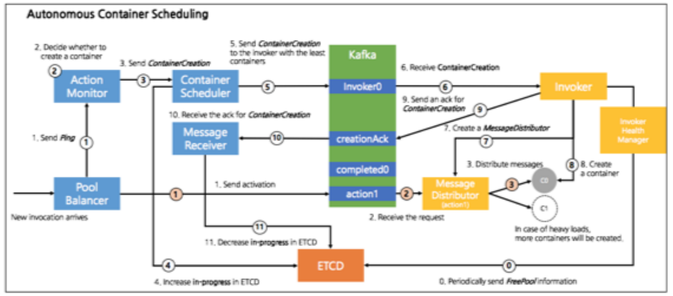
OW operators can choose the suitable scheduler which fits their needs the best.

In the ***Autonomous Container Scheduler***(I will call it ***ACS*** in the rest of this document), there are following major changes:

- Activation is handled in a best-effort manner.
  - One container will try its best to handle activations
  - When existing containers are not enough to properly handle incoming activations, more containers are created for the action.
- Once action containers are initialized, they exist longer than the current implementation.
  - They wait for subsequent requests for about 5~10s (configurable value).
- The criteria for throttling is no more ***activation per minute***, now the throttler cares ***the number of containers for the given action(namespace)*** and ***processing speed of existing containers***.
  - It denotes that once a container is initialized, it is occupied by the given action.

The following diagram depicts the basic flow of ***container creation*** and ***activation invocation***.

Each path is separately handled. So there are only 3 steps in the **activation path**.



System Architecture

Dominic Kim님이 작성, 4월 04, 2019에 최종 변경

Introduction

I introduced a few new components.

1) Scheduler

Scheduler is a new components it include many critical features.

It is in charge of two major features, 1. queueing and routing activations 2. decide whether to add more containers based on the loads.

It has sub component `Queue`. The role of the queue is similar to the Kafka topic. A dedicated queue is created for each action.

Each queue will receive activation messages from the Kafka for a given action and send them to the ContainerProxy in respond to requests from it.

2) etcd

etcd is a distributed reliable key-value store. etcd is mostly used for transactional support and information sharing among components.

3) Akka-grpc

Akka-grpc is introduced to replace Kafka based execution path.

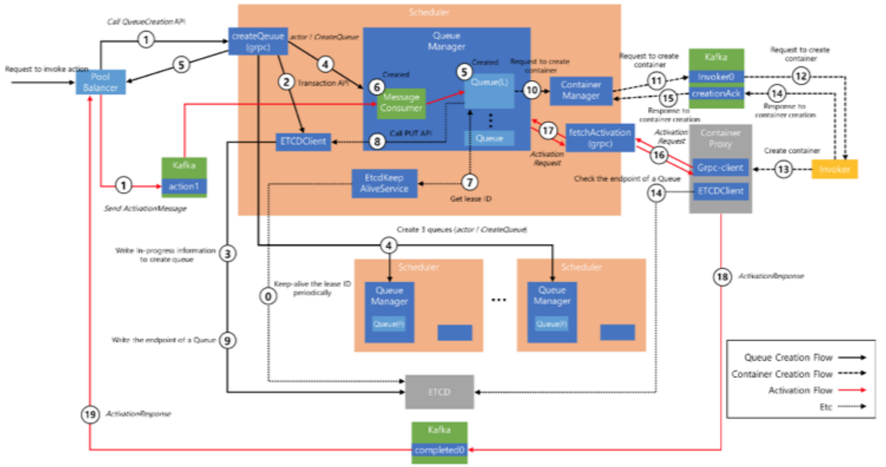
In this version, I could not fully rule out it due to heavy dependencies. My final objective is to exclude it at least from the critical path.

It is also being used to send queue creation request to the scheduler.

4) Akka-cluster

Akka-cluster is used for schedulers to communicate with each other.

Akka-grpc is required to define a grpc message, there are some advantages in akka-cluster when it is being used for simple inter-cluster communication.



Entire system is comprised of three flows, 1. queue creation flow, 2. container creation flow, 3. activation flow.

https://cwiki.apache.org/confluence/display/OPENWHISK/System+Architecture

신규 스케쥴러 구현

DEVIEW 2019

Distributed Scheduler #758

Merged **geonhui-kim** merged 92 commits into `master` from `without-debugger` on 4 Jun

Conversation 6 | Commits 92 | Checks 0 | Files changed 115

dominic-kim commented on 30 May

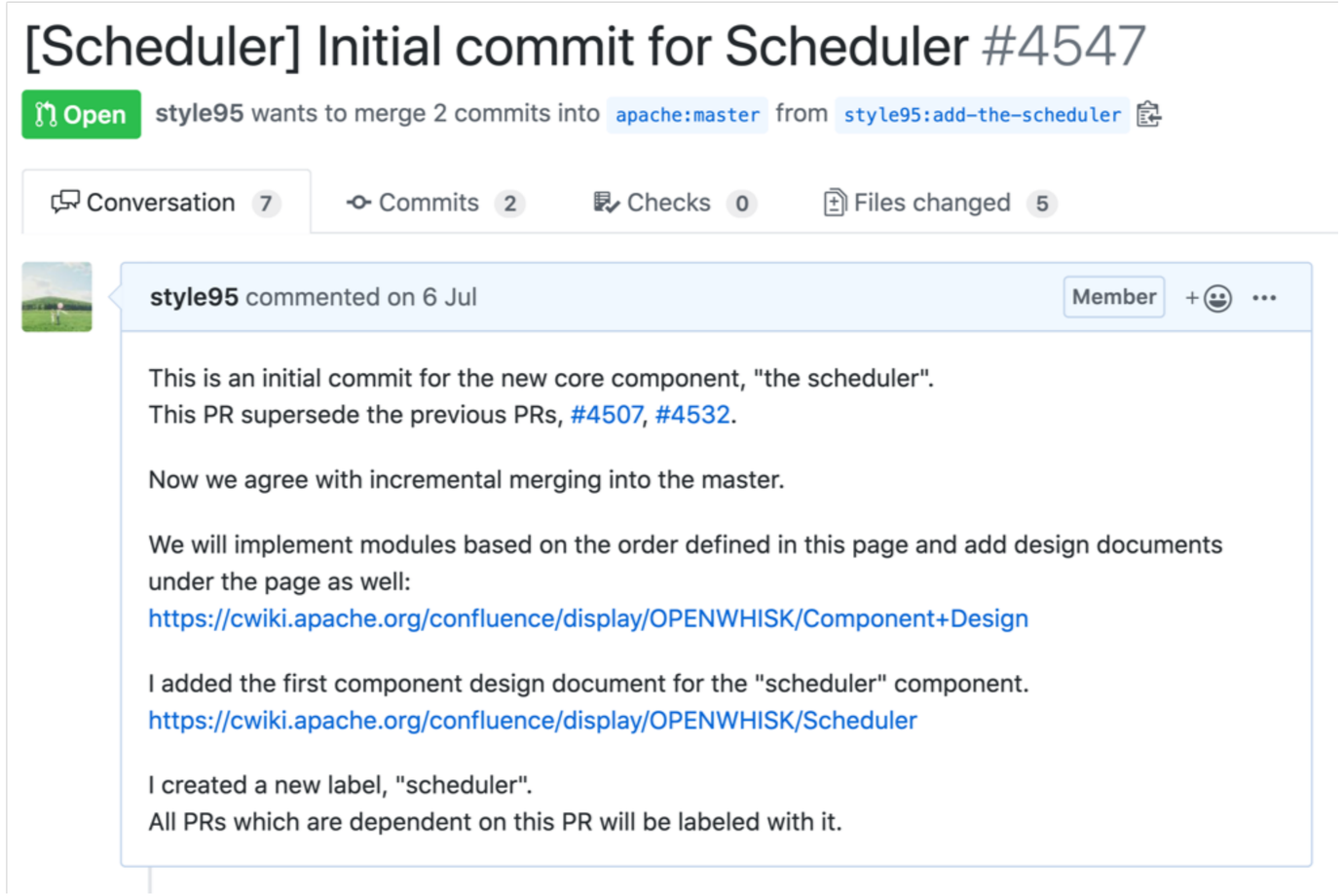
No description provided.

dominic-kim and others added some commits on 13 Feb

Initial commit for Scheduler	6580b39
Initial commit for Activation gRPC Service (#608)	ac8dd87
Fix unused-import issue (#610)	5e2e025
Etcd Utils (#609) ···	6e888e9
Add ActivationServiceImpl (#612) ···	4646177
Add Etcd Leadership api (#614)	5c6b89f
Unify Etcd clients (#615)	f73d044
Add MemoryQueue and leader election process (#616) ···	d6e3c98
Queue gRPC service (#617) ···	149221d

신규 스케쥴러 컨트리뷰션

DEVIEW
2019



Apache Committer/PMC 자격 획득

DEVIEW 2019

Invitation to become Apache OpenWhisk Committer and PPMC member: Dominic Kim

받은편지함 ×

Rodric Rabbah <rodric@gmail.com>
나, private에게

4월 6일 (토) 오후 8:51

Hello Dominic,

The Apache OpenWhisk Project Management Committee (PMC) hereby offers you committer privileges to the project as well as a membership on the Podling Project Mgmt. Committee (PPMC). Please read both parts of this email below.

Committer privileges are offered on the understanding that you'll use them reasonably and with common sense. We like to work on trust rather than unnecessary constraints.

Being a committer enables you to more easily make changes without needing to go through the patch submission process.

Being a committer does not require you to participate any more than you already do. It does tend to make one even more committed. You will probably find that you spend more time here.

Of course, you can decline and instead remain as a contributor, participating as you do now.

A. This personal invitation is a chance for you to accept or decline in private. Either way, please let us know in reply to the private@openwhisk.apache.org address only.

B. If you accept the invitation, you will also have to choose a unique Apache account username and let us know. We need this before we can proceed with account creation and assigning required permissions. To verify that your desired username does not already exist, please head over to the people index [1] and do a quick search to see if the login id is in use. Please also choose an email forwarding address for your Apache account.

요즘은

DEVIEW
2019

에러 및 로그가 제대로 출력되지 않는 이슈 #437

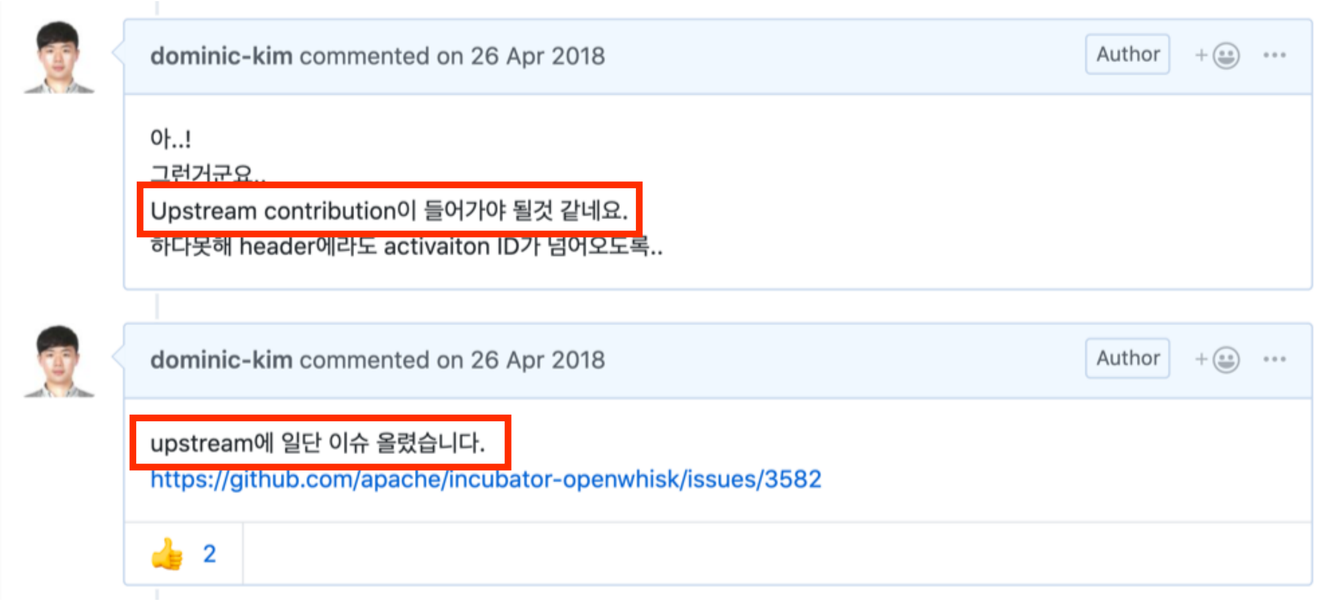
Closed **dominic-kim** opened this issue on 26 Apr 2018 · 15 comments

dominic-kim commented on 26 Apr 2018

아래의 액션을 실행해보면 결과와 로그창에 아래처럼 나옵니다.

요즘은

DEVIEW
2019



요즘은

DEVIEW 2019

[Feature Request] Include activation ID in web action header. #3582

Edit | New issue

Closed **style95** opened this issue on 26 Apr 2018 · 4 comments · Fixed by #3671

style95 commented on 26 Apr 2018 — Member

Environment details:

- Any

Steps to reproduce the issue:

1. Invoke a web action.

Provide the expected results and outputs:

```
function main(params) {
    return { response: params };
}
```

```
curl -X GET 'localhost/api/v1/web/myNamespace/hello.json'
{
  "response": {
    "__ow_method": "get",
```

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Notifications Customize

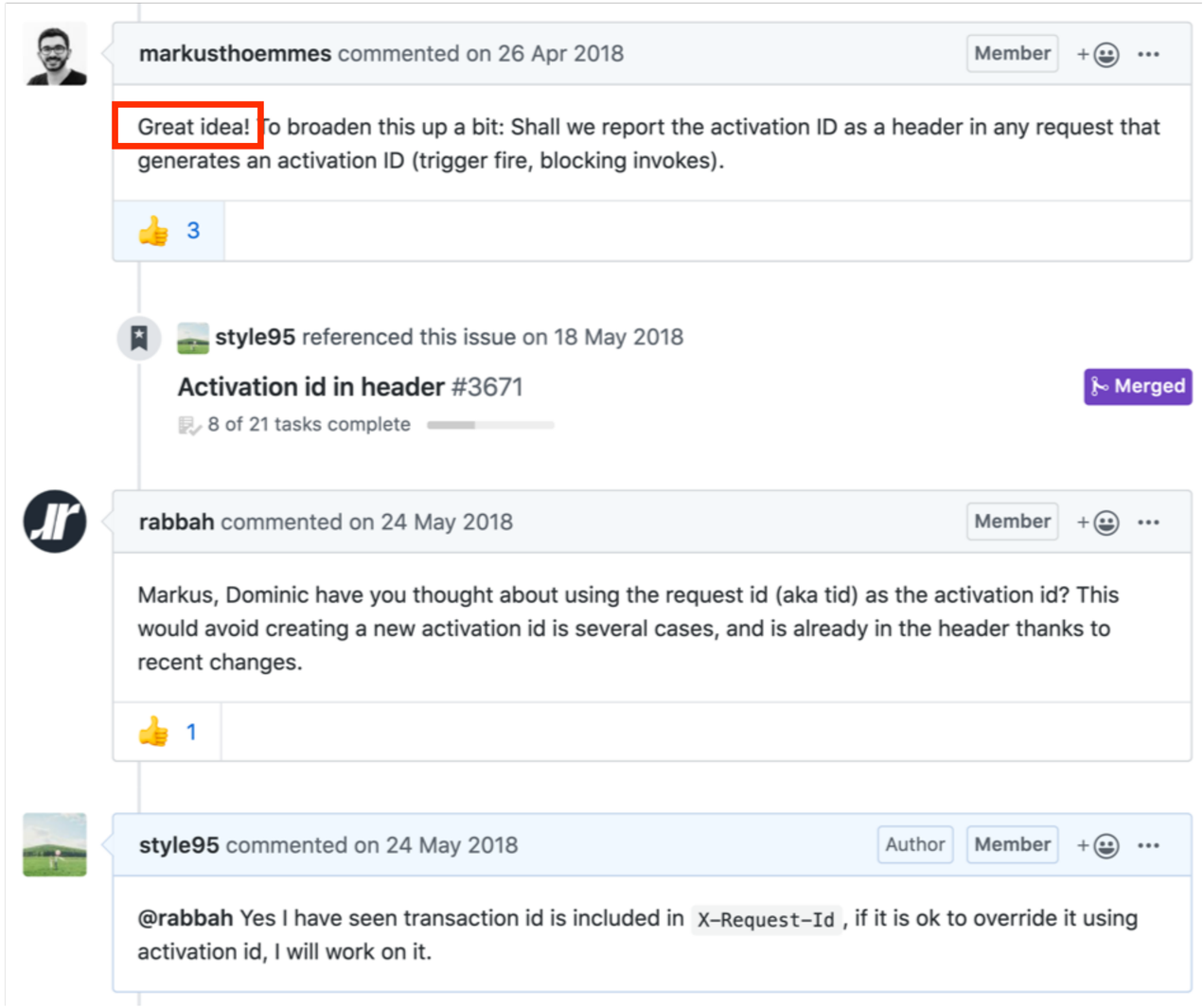
Unsubscribe

You're receiving notifications because you're watching this repository.

3 participants

요즘은

DEVIEW 2019



markusthoemmes commented on 26 Apr 2018 — Member

Great idea! To broaden this up a bit: Shall we report the activation ID as a header in any request that generates an activation ID (trigger fire, blocking invokes).

3

style95 referenced this issue on 18 May 2018

Activation id in header #3671 — Merged

8 of 21 tasks complete

rabbah commented on 24 May 2018 — Member

Markus, Dominic have you thought about using the request id (aka tid) as the activation id? This would avoid creating a new activation id is several cases, and is already in the header thanks to recent changes.

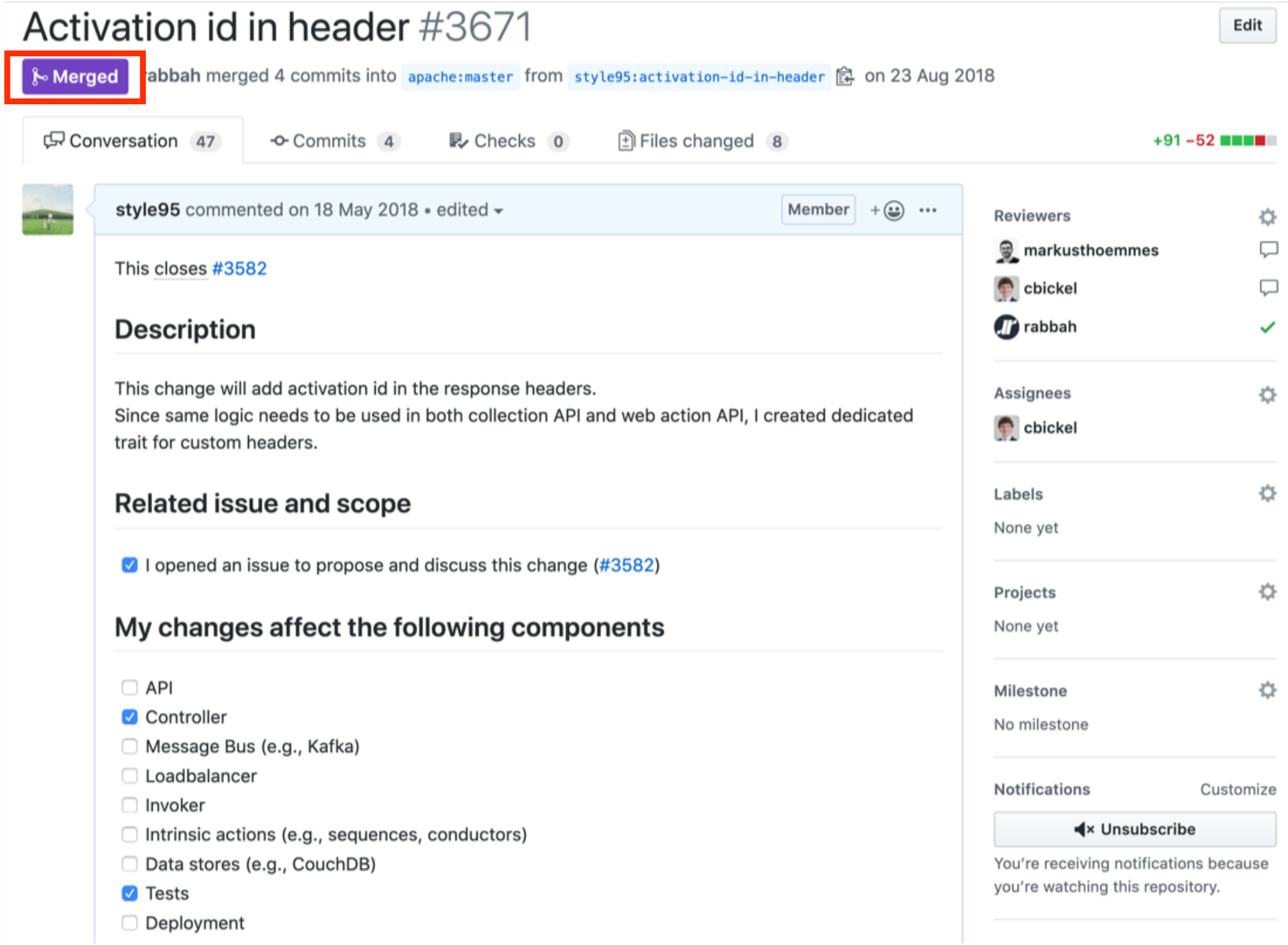
1

style95 commented on 24 May 2018 — Author, Member

@rabbah Yes I have seen transaction id is included in `X-Request-Id`, if it is ok to override it using activation id, I will work on it.

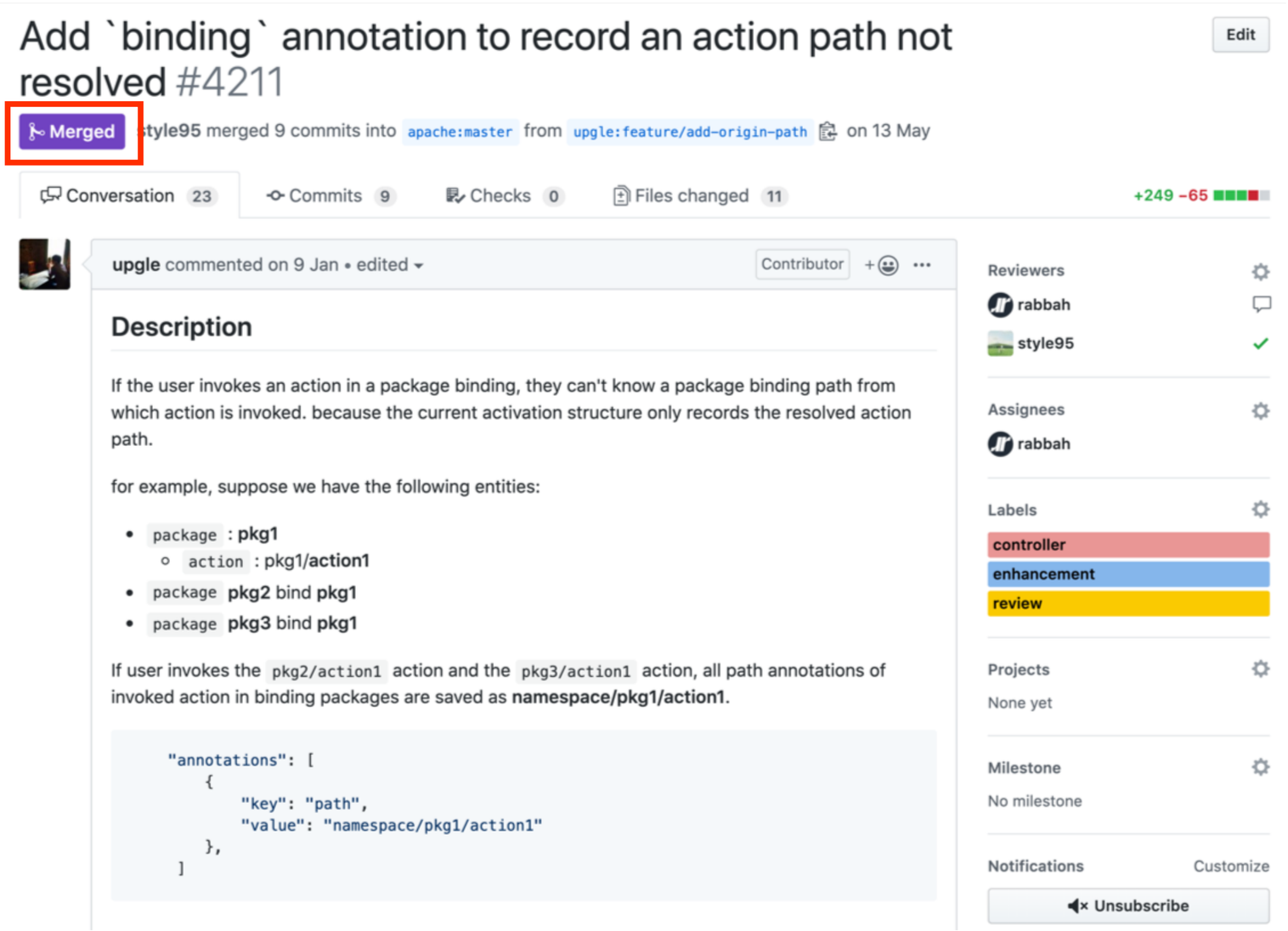
요즘은

DEVIEW 2019



요즘은

DEVIEW 2019



Add `binding` annotation to record an action path not resolved #4211

Edit

Merged style95 merged 9 commits into apache:master from upgle:feature/add-origin-path on 13 May

Conversation 23 Commits 9 Checks 0 Files changed 11 +249 −65

upgle commented on 9 Jan • edited Contributor

Description

If the user invokes an action in a package binding, they can't know a package binding path from which action is invoked. because the current activation structure only records the resolved action path.

for example, suppose we have the following entities:

- `package` : **pkg1**
  - `action` : pkg1/**action1**
- `package` **pkg2** bind **pkg1**
- `package` **pkg3** bind **pkg1**

If user invokes the `pkg2/action1` action and the `pkg3/action1` action, all path annotations of invoked action in binding packages are saved as **namespace/pkg1/action1**.

```
"annotations": [
    {
        "key": "path",
        "value": "namespace/pkg1/action1"
    },
]
```

Reviewers
rabbah
style95

Assignees
rabbah

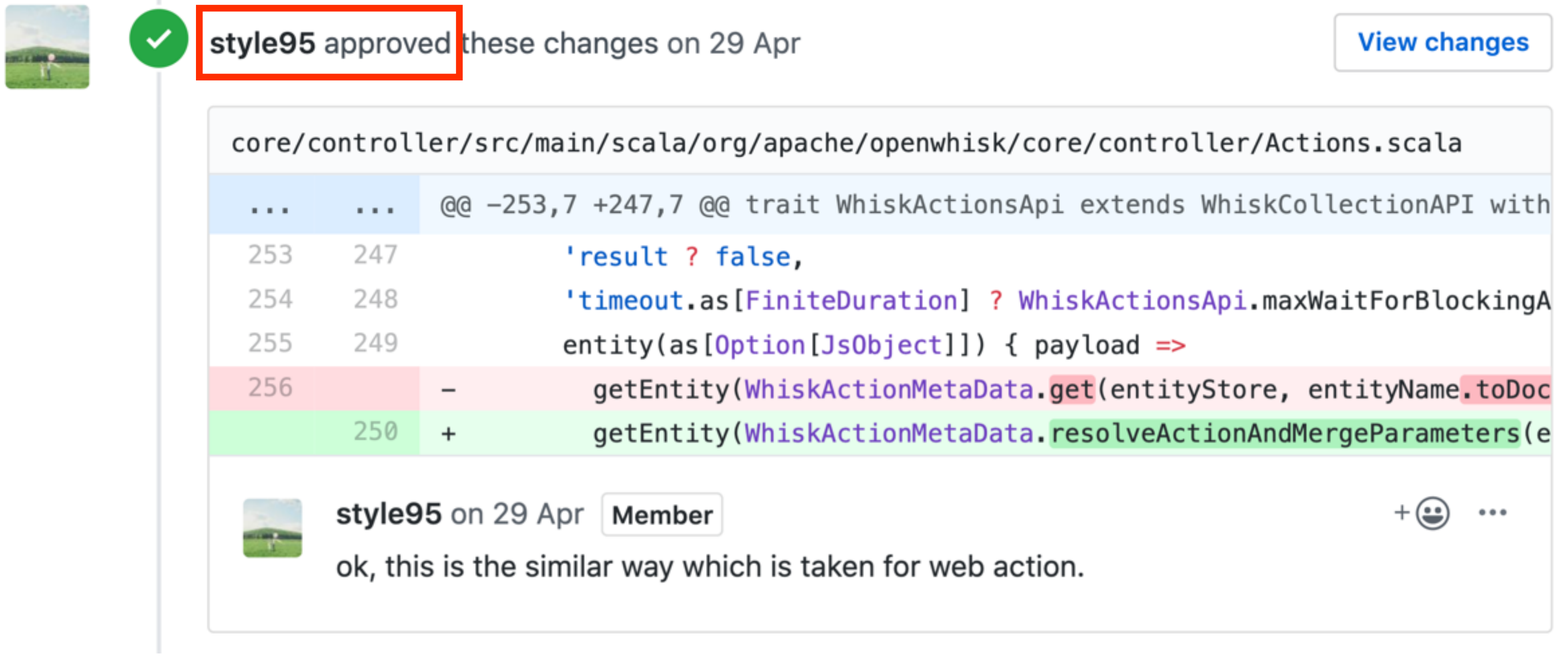
Labels
controller
enhancement
review

Projects
None yet

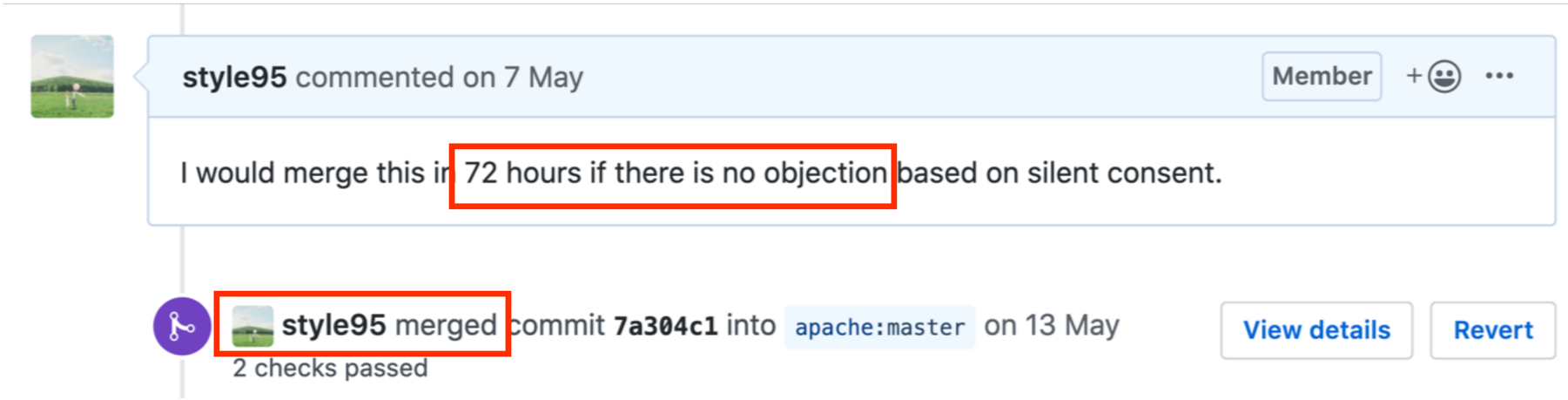
Milestone
No milestone

Notifications Customize
Unsubscribe

요즘은

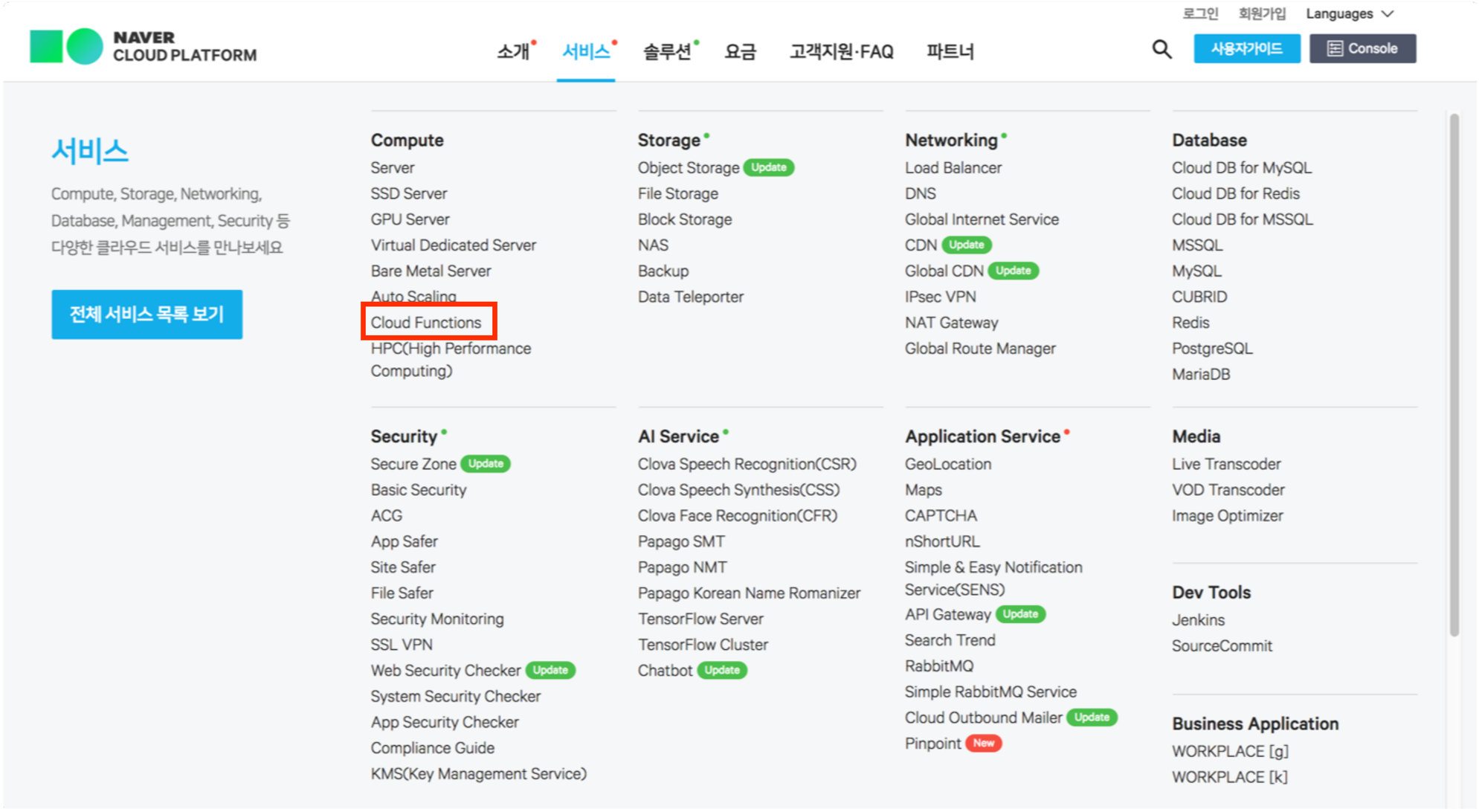

style95 approved these changes on 29 Apr
View changes
core/controller/src/main/scala/org/apache/openwhisk/core/controller/Actions.scala
... ... @@ -253,7 +247,7 @@ trait WhiskActionsApi extends WhiskCollectionAPI with
253 247 'result ? false,
254 248 'timeout.as[FiniteDuration] ? WhiskActionsApi.maxWaitForBlockingA
255 249 entity(as[Option[JsObject]]) { payload =>
256 - getEntity(WhiskActionMetaData.get(entityStore, entityName.toDoc
250 + getEntity(WhiskActionMetaData.resolveActionAndMergeParameters(e
style95 on 29 Apr
Member
ok, this is the similar way which is taken for web action.

요즘은


style95 commented on 7 May
Member
I would merge this in 72 hours if there is no objection based on silent consent.
style95 merged commit 7a304c1 into apache:master on 13 May
2 checks passed
View details
Revert

요즘은

DEVIEW 2019



DEVIEW
2019

Q & A

DEVIEW
2019

Thank You