

OpenWhisk Scheduling Proposal

Terms and Notions

- *ActionN*: the action whose name is N, also implies activation request for the *ActionN*
- *ExecutionN*: imply the execution of *ActionN*
- *WarmedN*: warmed container of *ActionN*
- *controllerM*: the controller whose index is M
- *completedM*: Kafka topic for *controllerM* to receive completion message
- *invokerN*: the invoker whose index is N, also it implies Kafka topic for *invokerN*
- *homeInvoker*: a target invoker to which the given action would be scheduled

Current Implementation Details

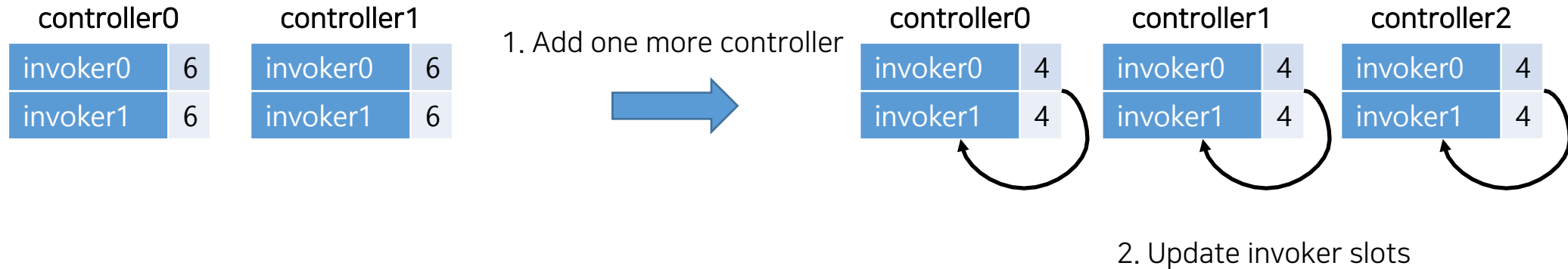
Index

1. InvokerSlot
2. Hash function, homeInvoker, stepSize
3. Forcable Semaphore
4. ContainerPool
5. ContainerProxy

Current Implementation (1/6) – *Invoker Slot*

- Invoker has *MaxPoolSize(numCore * coreShare)*
- Each controllers has its own invoker slots
 - It is semi-proportional to the number of controllers (*invokerN slot in a controller = MaxPoolSize / # of controllers*)
 - It is dynamically changed as controllers join/leave the cluster.

※ Invoker MaxPoolSize: 12



Current Implementation (2/6) – *Hash function, HomeInvoker, StepSize*

- Loadbalancer use *hash function* to decide *homeInvoker* of a given action

```
def hashFunction(namespace, action) = {  
    (namespace.asString.hashCode() ^ action.asString.hashCode()).abs  
    homeInvoker = hash % invokersToUse.size  
    return homeInvoker  
}
```

- Loadbalancer use *stepSize* to choose different invoker in case *homeInvoker* is not available
 - *stepSize* is the number which is coprime with the number of invokers
 - For more about *stepSize*: <https://github.com/apache/incubator-openwhisk/pull/2360>

$[newIndex = (oldIndex + step) \% numInvokers]$

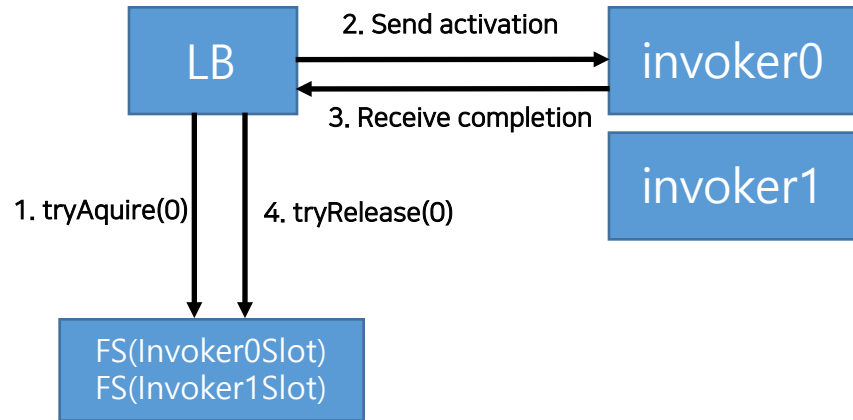
[Example]

- $oldIndex = 0, numInvokers = 3$ (0,1,2)
- coprime number with 3 -> 5
- $newIndex1 = (0 + 5) \% 3 \rightarrow 2$
- $newIndex2 = (2 + 5) \% 3 \rightarrow 1$
- $newIndex3 = (1 + 5) \% 3 \rightarrow 0$
- Can iterate all invokers

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

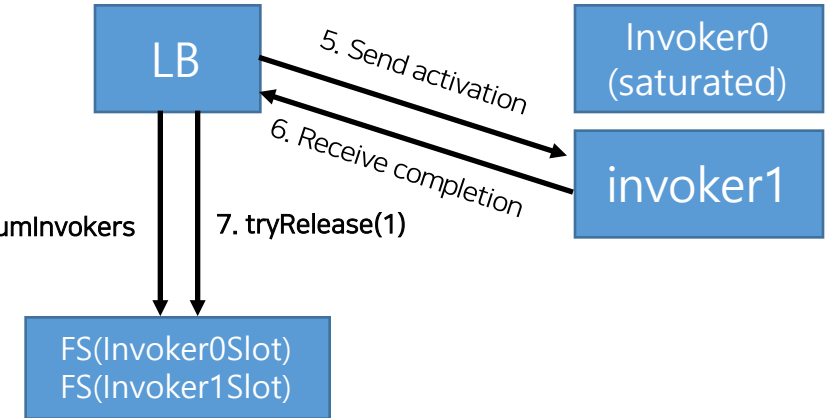
- Loadbalancer chooses invoker based on *ForcableSemaphore*

Normal situation



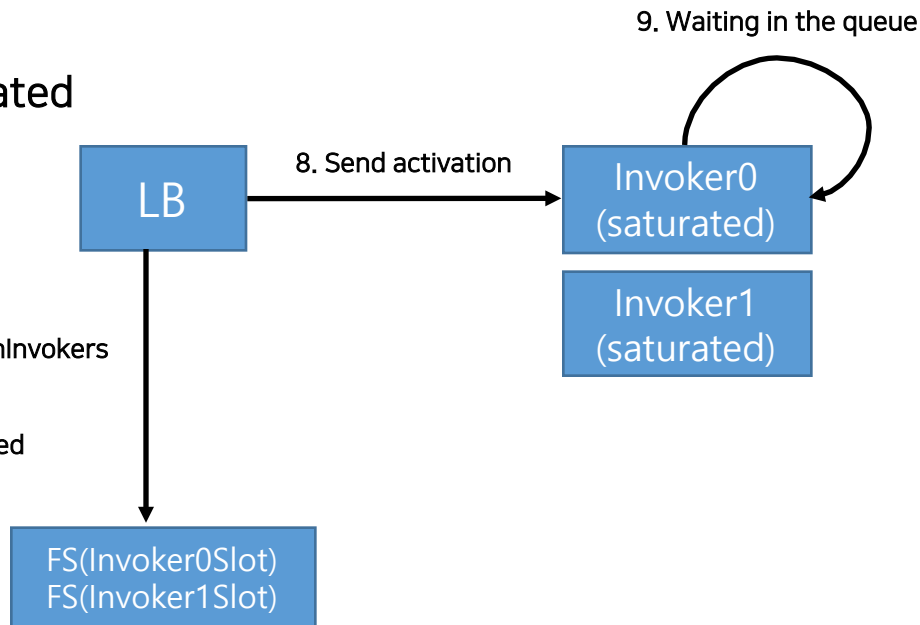
Home Invoker is saturated

1. tryAquire(0)
2. Failed to acquire
3. $newIndex = (0 + step) \% numInvokers$
4. tryAquire(1)



All invokers are saturated

1. tryAquire(0)
2. Failed to acquire
3. $newIndex = (0 + step) \% numInvokers$
4. tryAquire(1)
5. Failed to acquire
6. $newIndex = randomlyChosed$
7. **forceAcquire(0)**



Current Implementation (4/6) – *ContainerPool*

- Invoker has *ContainerPool* to keep status of containers.
- *ContainerPool* keeps status of 3 pools(*freePool*, *busyPool*, *prewarmPool*).

```
var freePool = immutable.Map.empty[ActorRef, ContainerData]
var busyPool = immutable.Map.empty[ActorRef, ContainerData]
var prewarmedPool = immutable.Map.empty[ActorRef, ContainerData]
```

- When activation message comes, first, it will try to find *warmed container* from *freePool*.
- If no warmed container found, it checks current pool size and take *PrewarmContainer* or create new one(*ColdStart*).
- If (*busyPool.size* + *freePool.size*) is greater than or equals to *maxPoolSize*, it will try to remove a container from *freePool* and take *PrewarmContainer* or create new one(*ColdStart*). If no container is deletable, just returns *None*.

```
if (busyPool.size + freePool.size < maxPoolSize) {
```

```
    ContainerPool.remove(freePool).map { toDelete =>
        removeContainer(toDelete)
        takePrewarmContainer(r.action)
        .map(container => {
            (container, "recreated")
        })
        .getOrElse {
            (createContainer(), "recreated")
        }
    }
}
```

Current Implementation (5/6) – *ContainerPool cont`*

- If any containers found, or created, invokers send activation message to it and removes it from *freePool* and add it into *busyPool*.

```
case Some(((actor, data), containerState)) =>
  busyPool = busyPool + (actor -> data)
  freePool = freePool - actor
  actor ! r // forwards the run request to the container
  logContainerStart(r, containerState)
```

- If *None* is returned(no container available), it reschedules that message to itself for at most 10 seconds.

```
self ! Run(r.action, r.msg, retryLogDeadline)
```


Current Implementation (6/6) – *ContainerProxy*

- *ContainerProxy* is logical container representative in invoker side. It is implemented based on *Akka[FSM]*

- It has many states.

case object Uninitialized **extends** ContainerState -> transient state to **trigger Prewarm container creation**. Next: *Starting*.

case object Starting **extends** ContainerState -> transient state to **wait until Prewarm container is created**. It would be registered in *PreWarmPool*. Next: *Started*.

case object Started **extends** ContainerState -> it is ready to receive a job. Once job message is received, it **initializes the container and run the code**. Next: *Running*

case object Running **extends** ContainerState -> transient state to **wait until code execution is finished**. Once it receives result, it is registered in *WarmPool*. Next: *Ready*

case object Ready **extends** ContainerState -> transient state to wait for subsequence run requests. Only **wait for 50ms and pause the container**. Next: *Pausing*

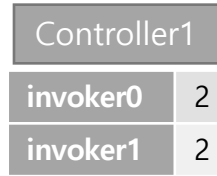
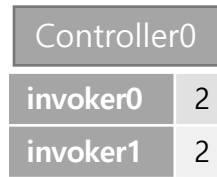
case object Pausing **extends** ContainerState -> transient state to **wait until container is paused**. Next: *Paused*

case object Paused **extends** ContainerState -> **wait until job request comes**. Once job request comes, it is resumed, run the code and move to *Running* state again. If no job request comes for **10 minutes**. It will destroy the container and move to *Removing*. Next: *Running* or *Removing*

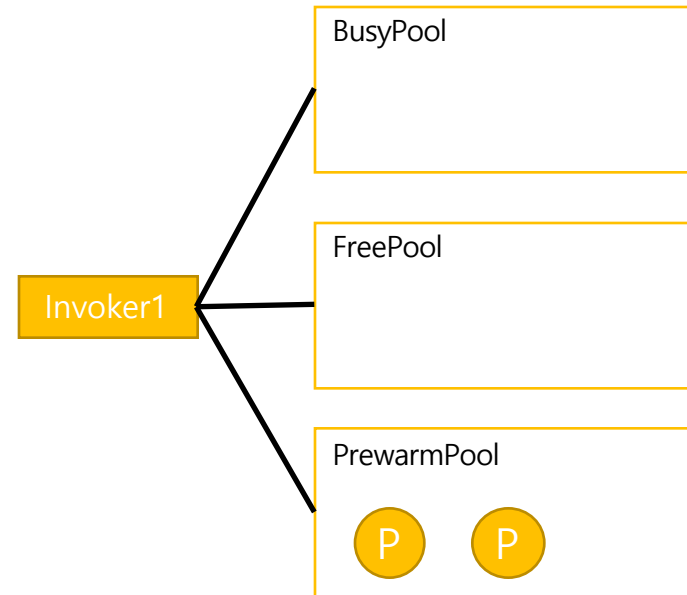
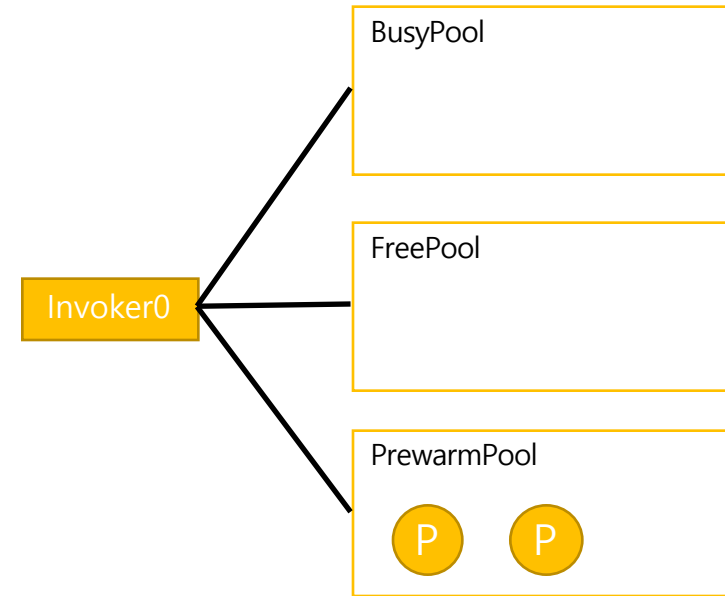
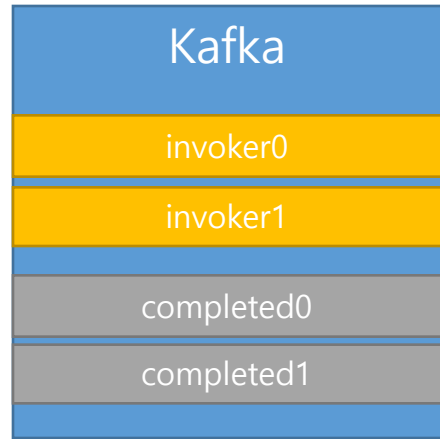
case object Removing **extends** ContainerState -> transient state to **wait until the container is removed**. If container is removed, FSM is destroyed as well.

- When running codes, it subsequently calls `/init`` and `/run`` REST API against the container.
 - Once code execution is over, it send completion message to controller(*completedM*).
 - And collect logs from the container and store it in CouchDB.

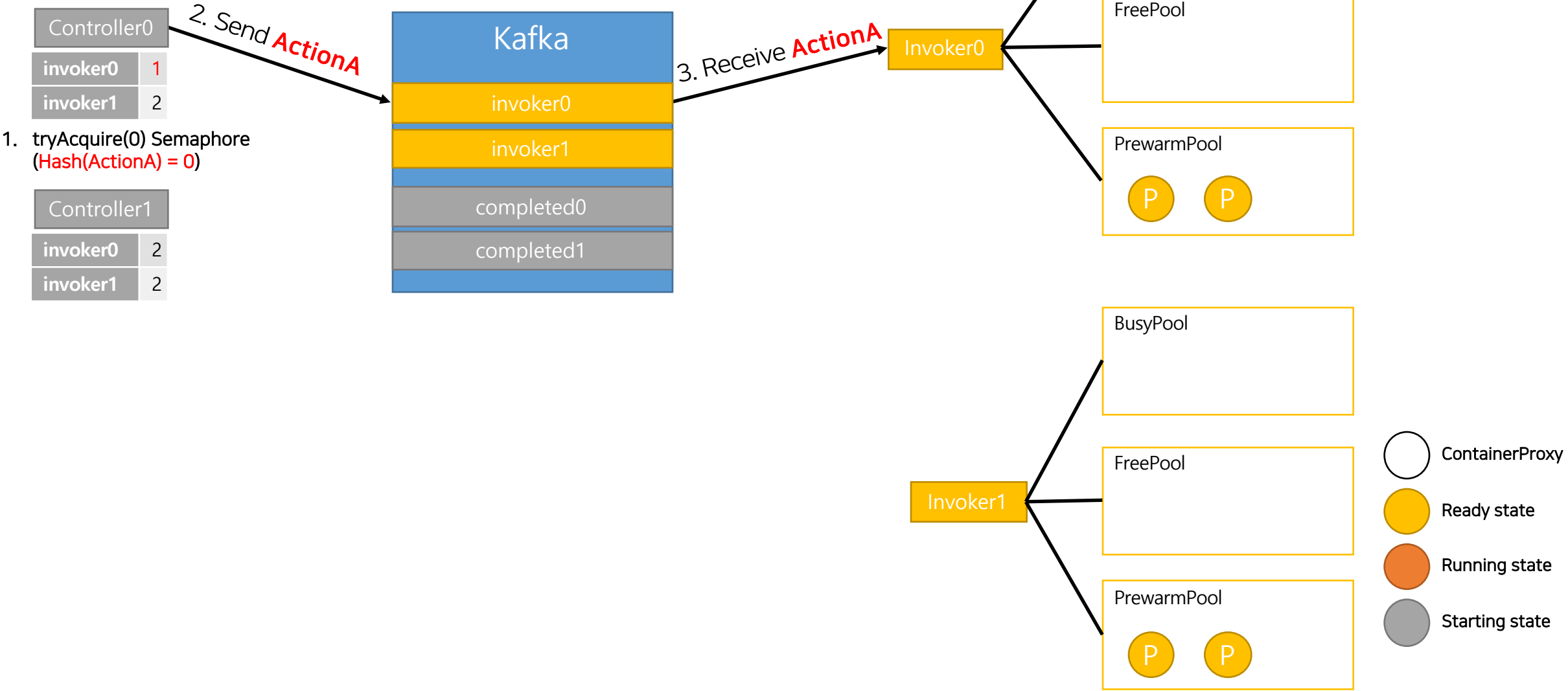
Basic Flow: What happens in the real scene



Forcible Semaphore



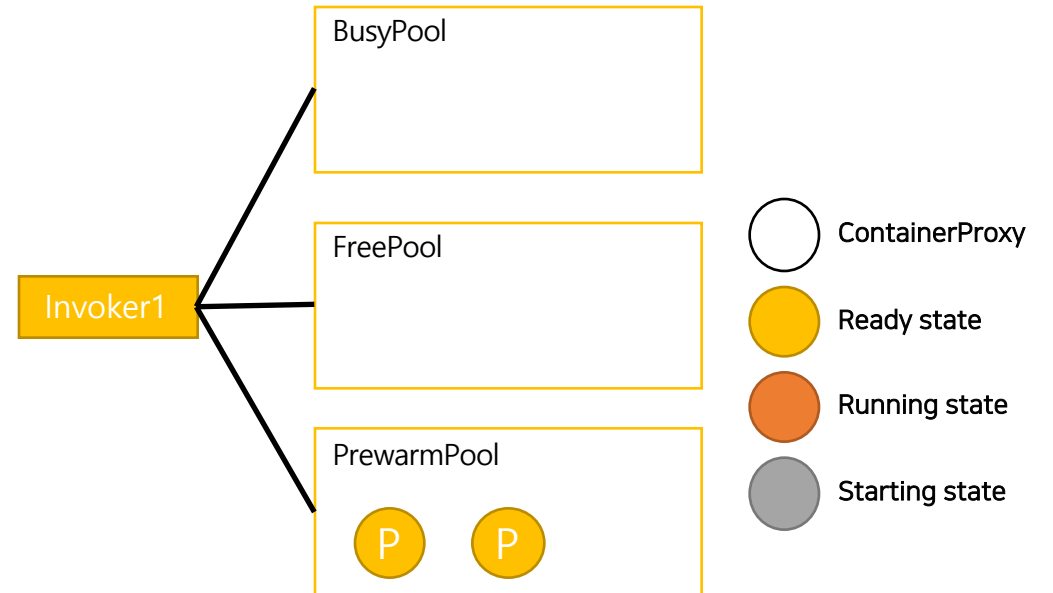
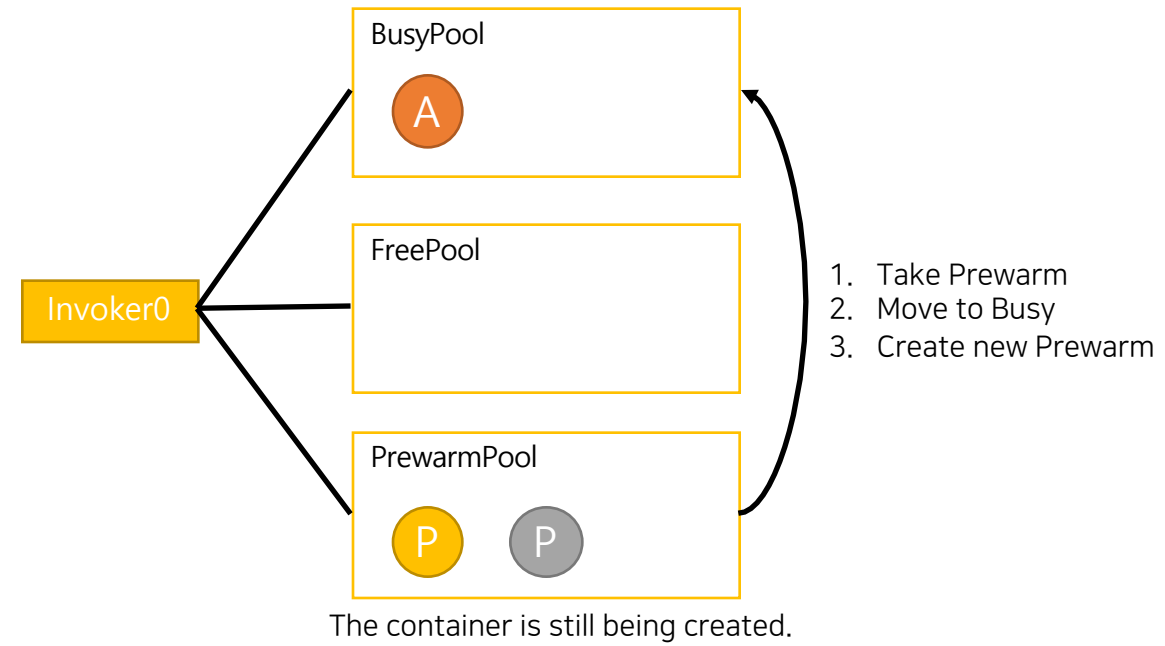
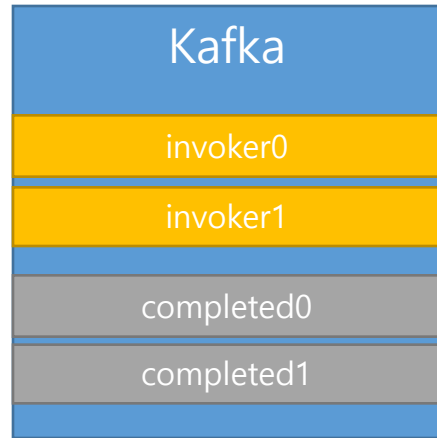
Basic Flow: What happens in the real scene



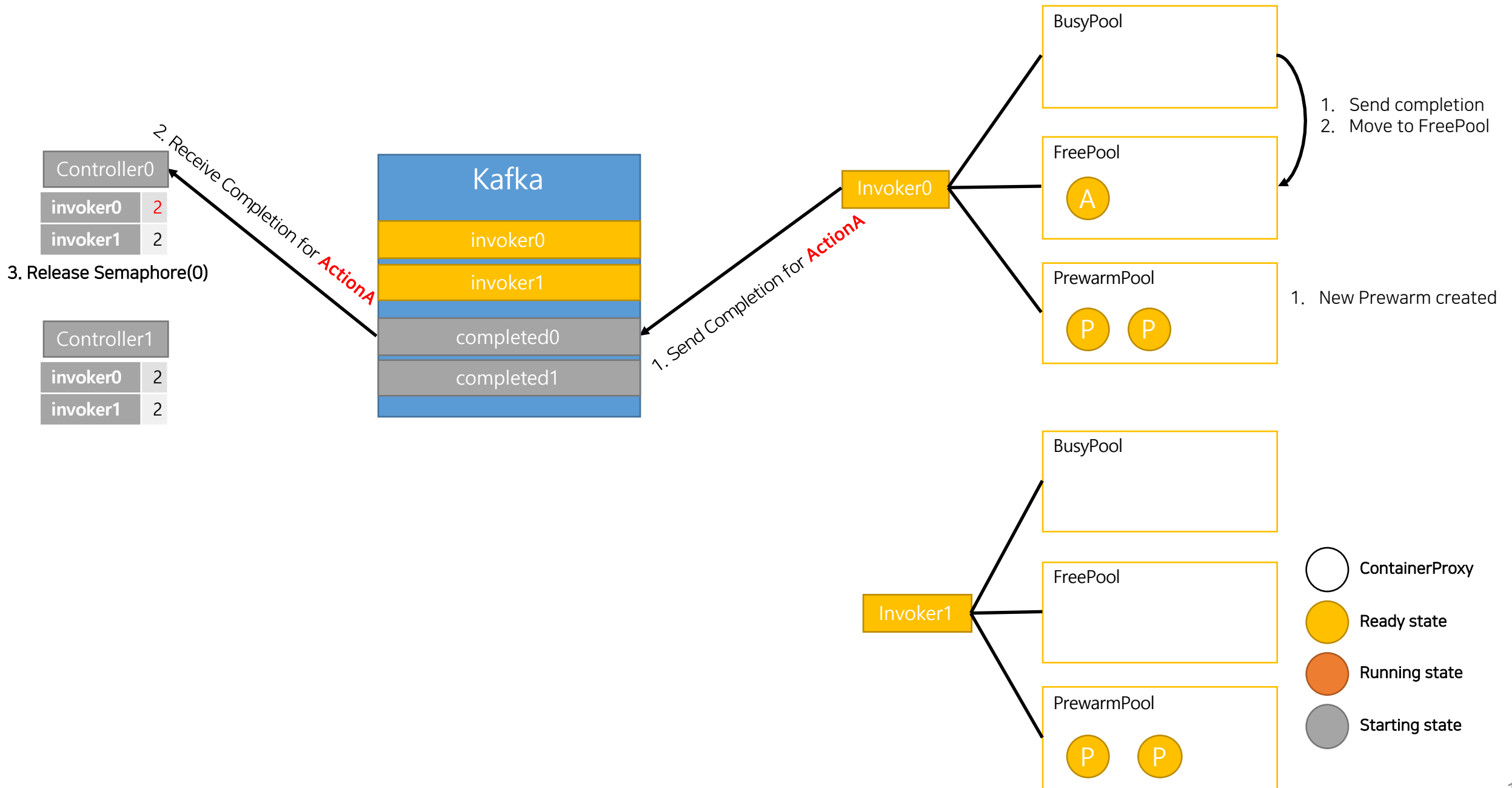
Basic Flow: What happens in the real scene

Controller0	
invoker0	1
invoker1	2

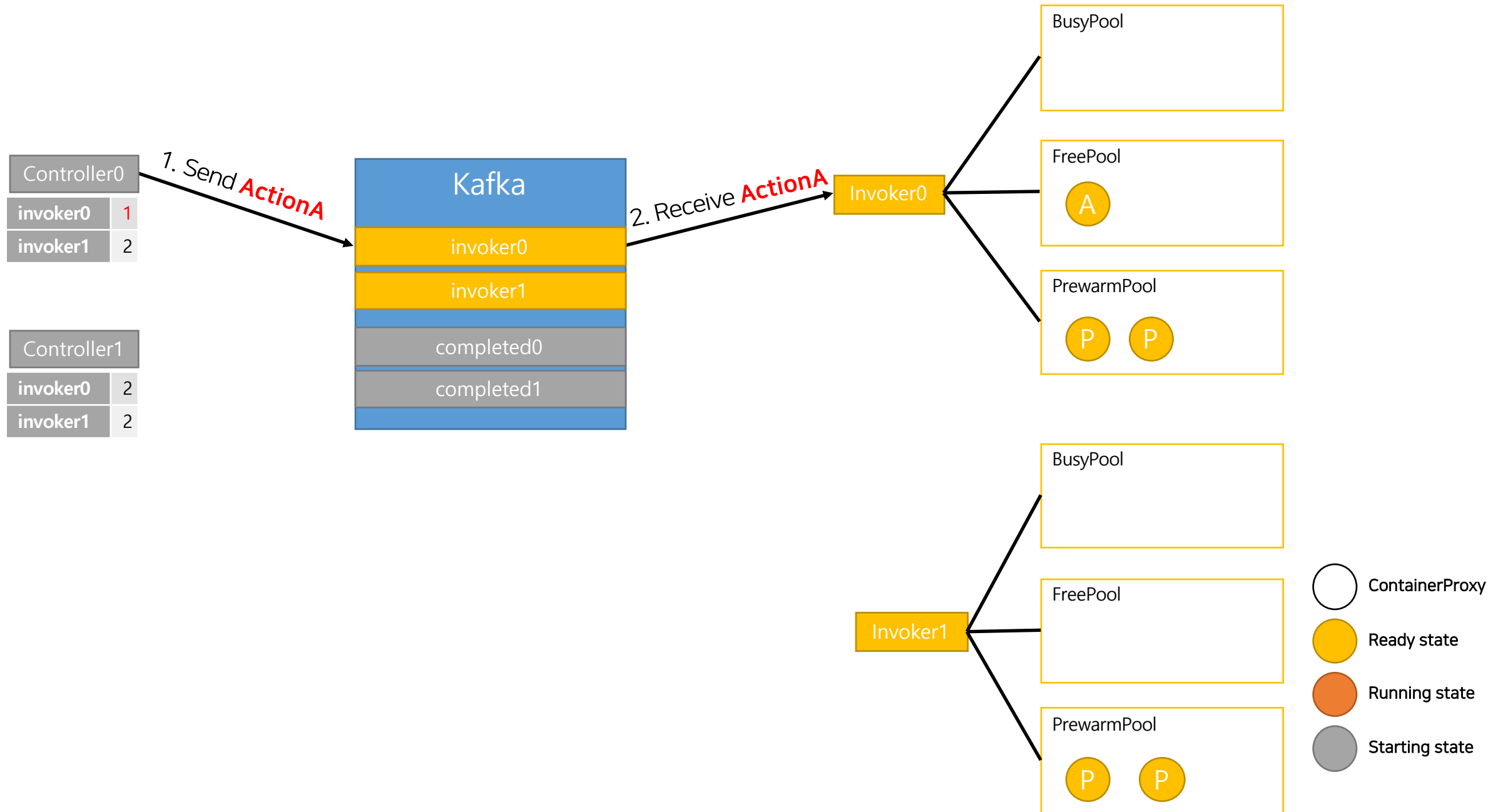
Controller1	
invoker0	2
invoker1	2



Basic Flow: What happens in the real scene



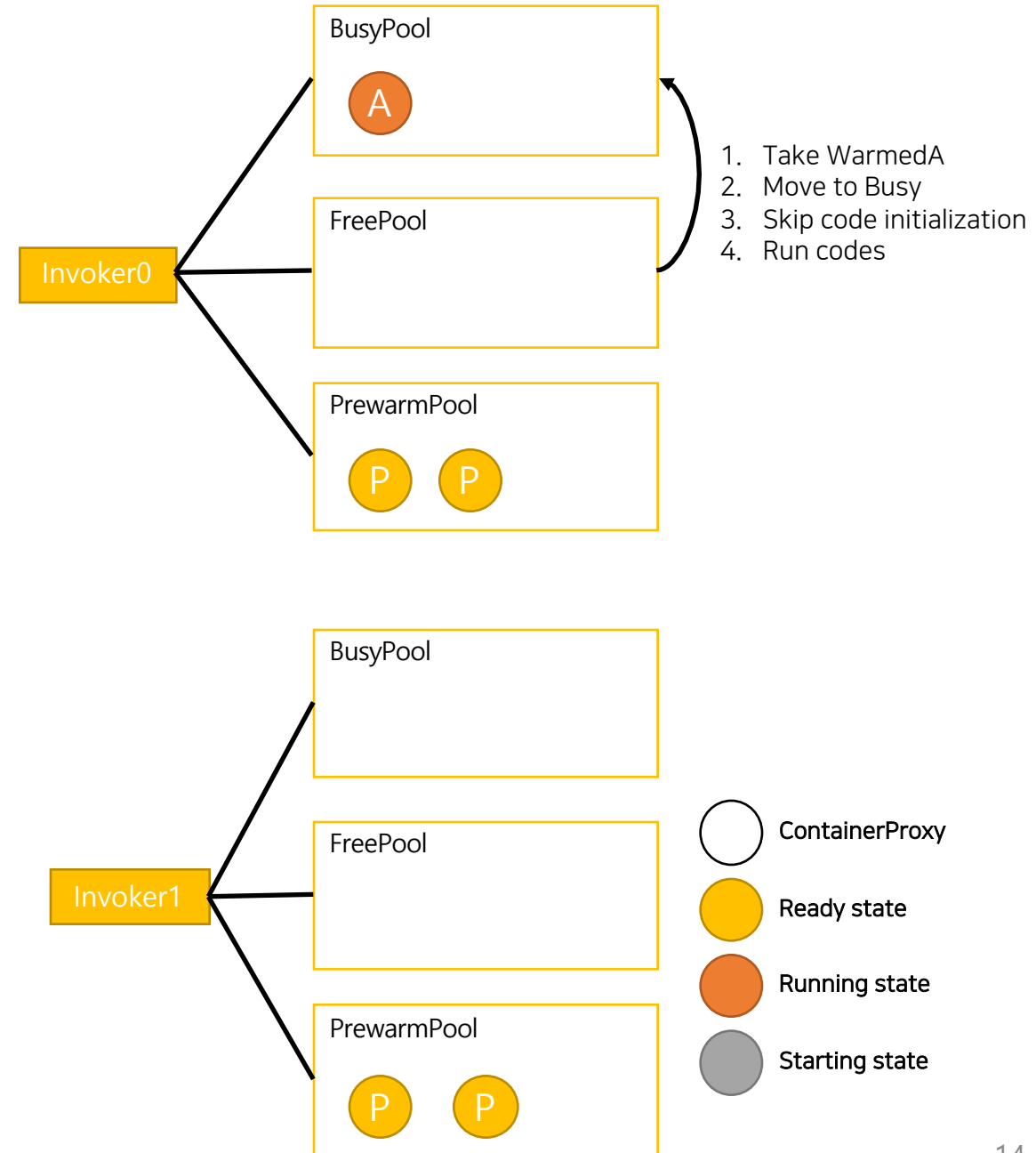
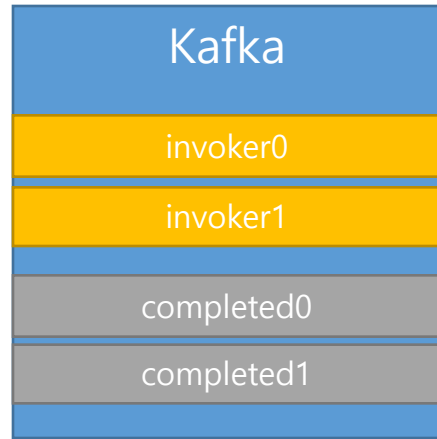
Warmed Flow: What happens in the real scene



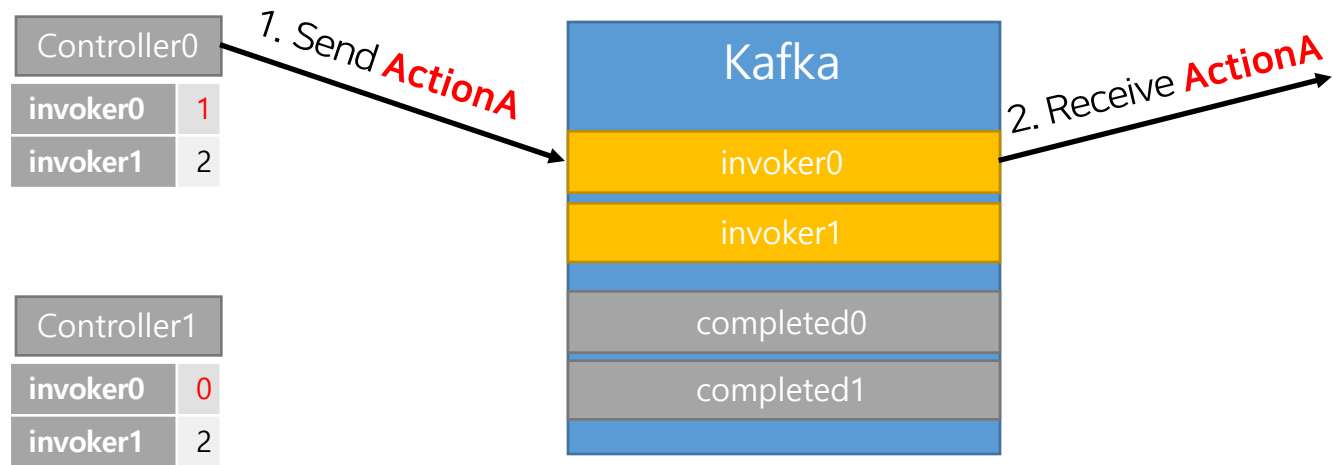
Warmed Flow: What happens in the real scene

Controller0	
invoker0	1
invoker1	2

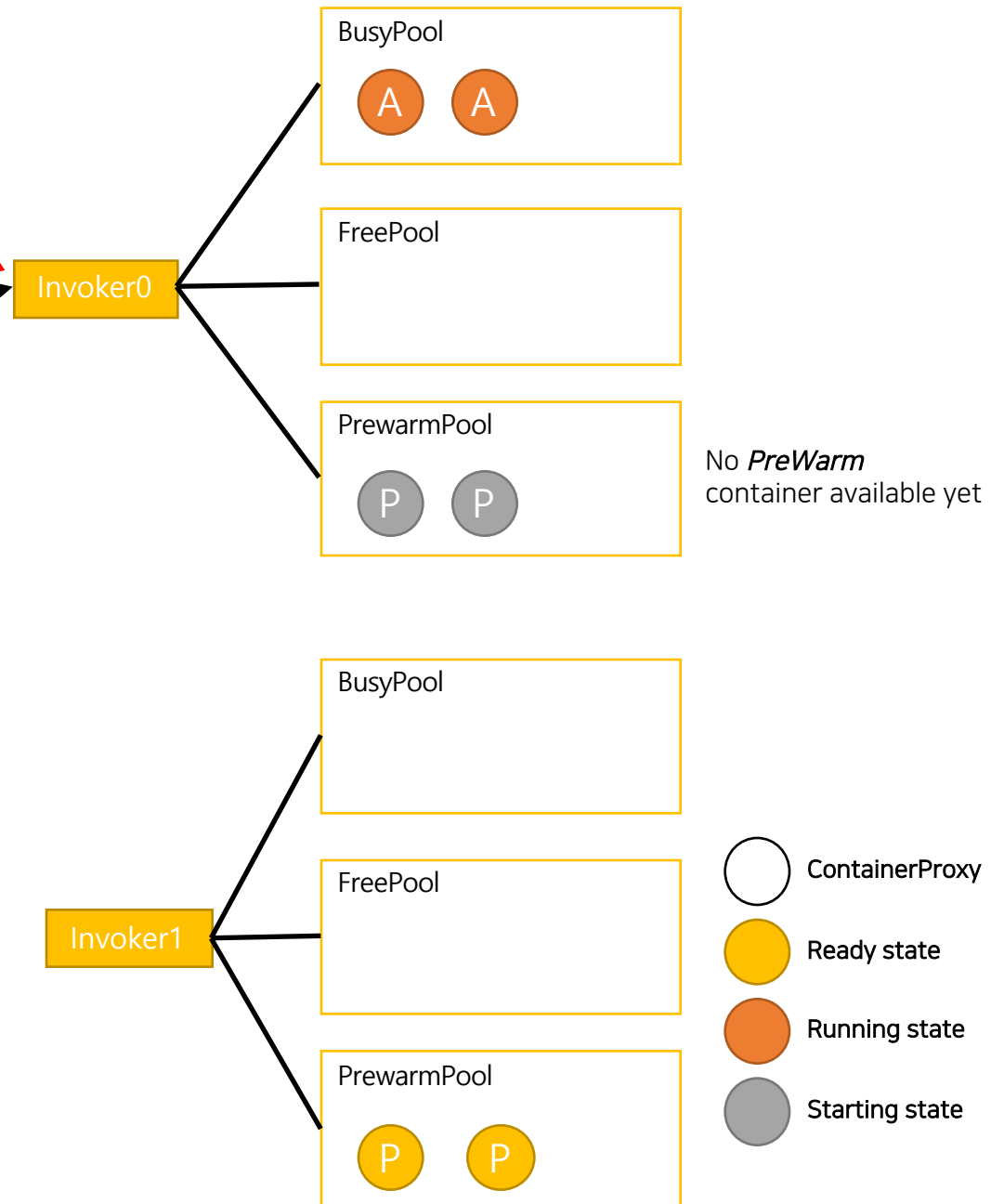
Controller1	
invoker0	2
invoker1	2



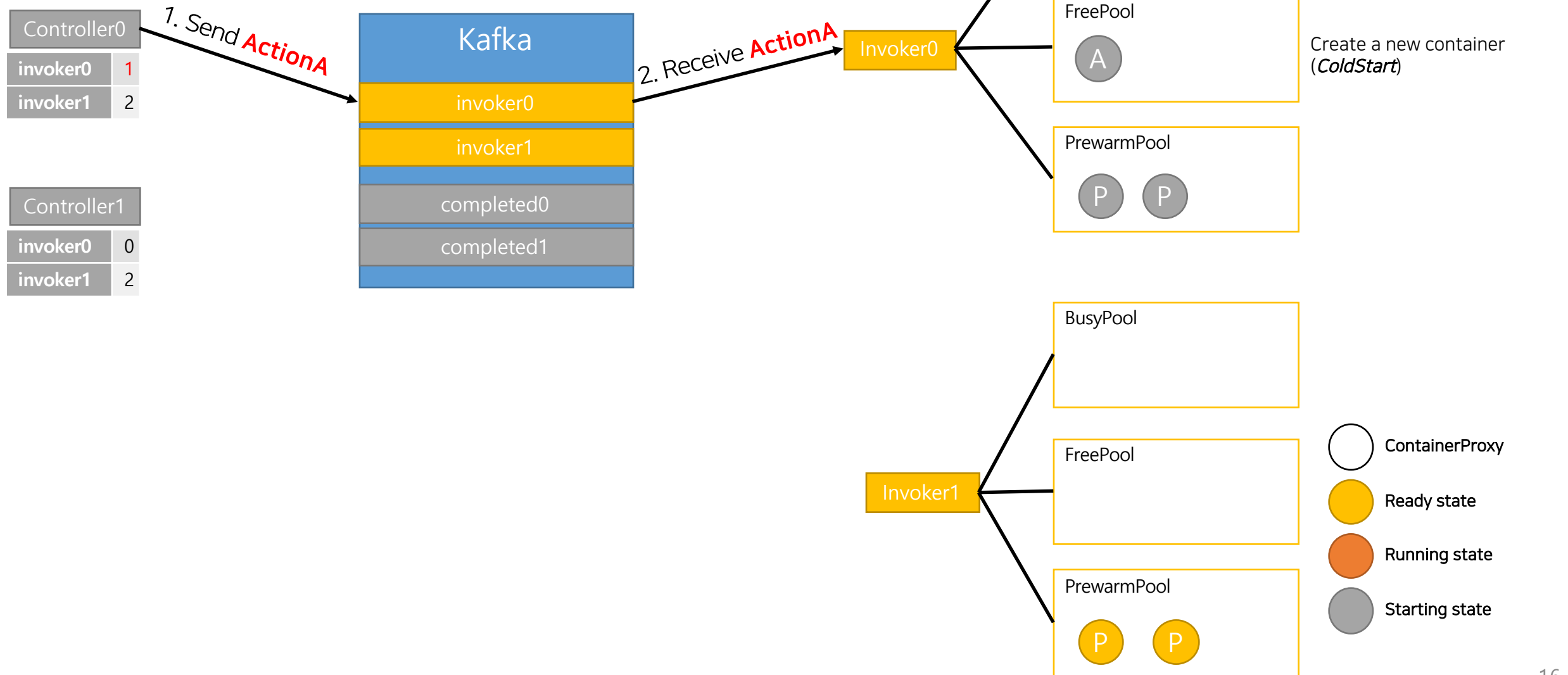
ColdStart Flow: What happens in the real scene



0. Two previous requests already took all Prewarm containers.



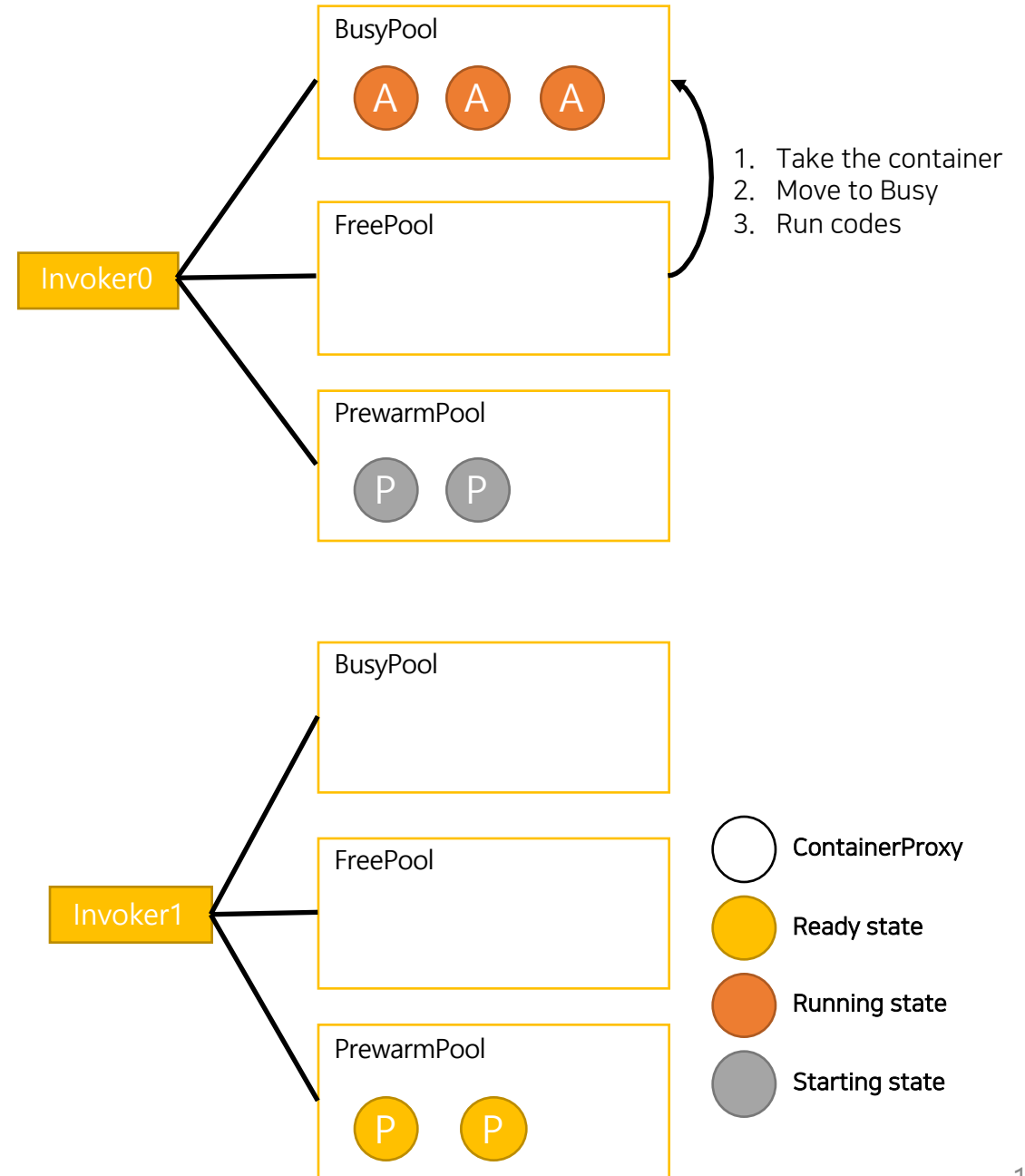
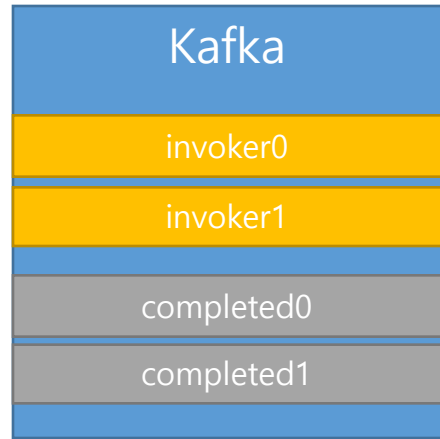
Cold Start Flow: What happens in the real scene



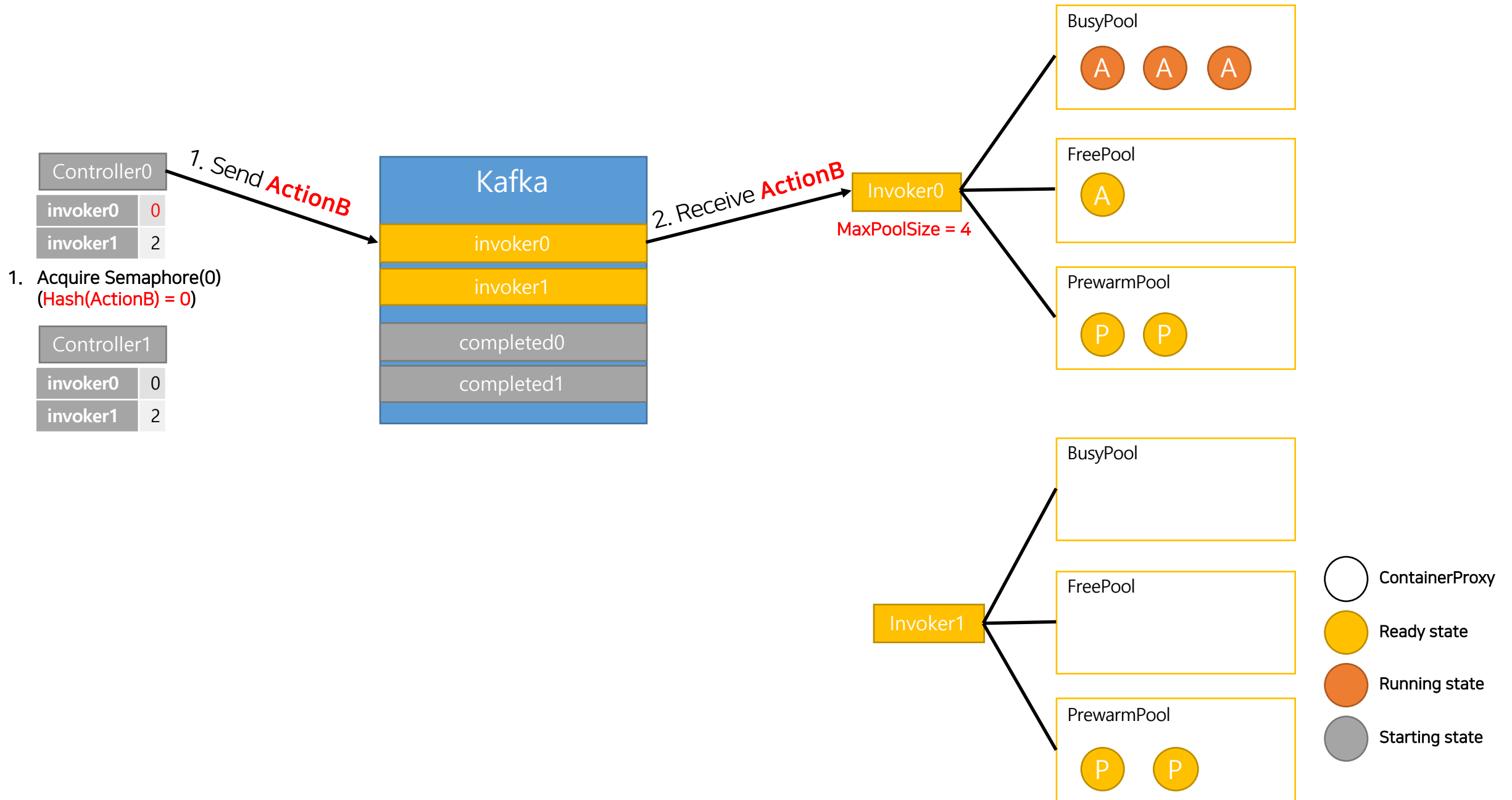
Cold Start Flow: What happens in the real scene

Controller0	
invoker0	1
invoker1	2

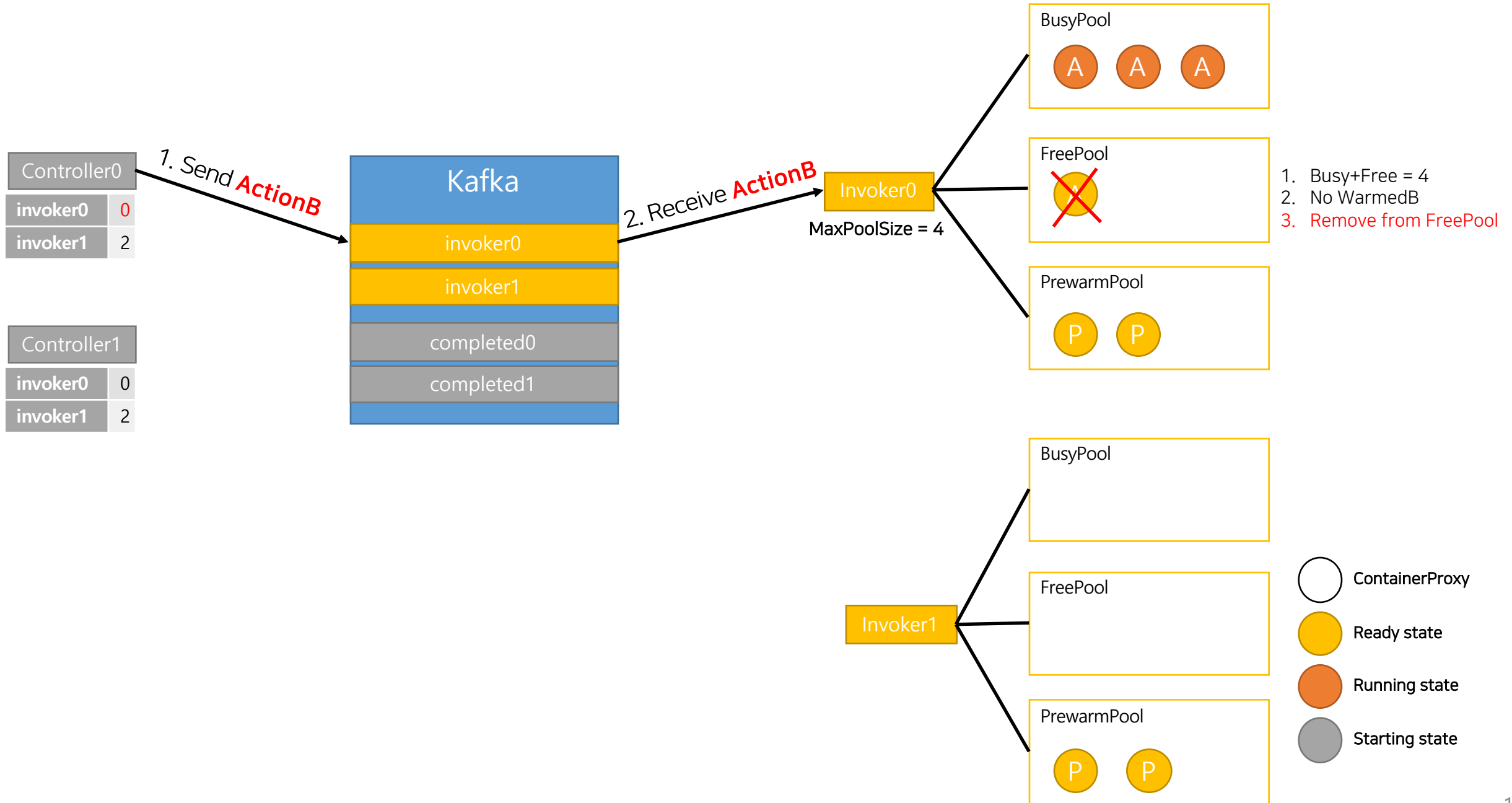
Controller1	
invoker0	0
invoker1	2



Container Deletion Flow: What happens in the real scene



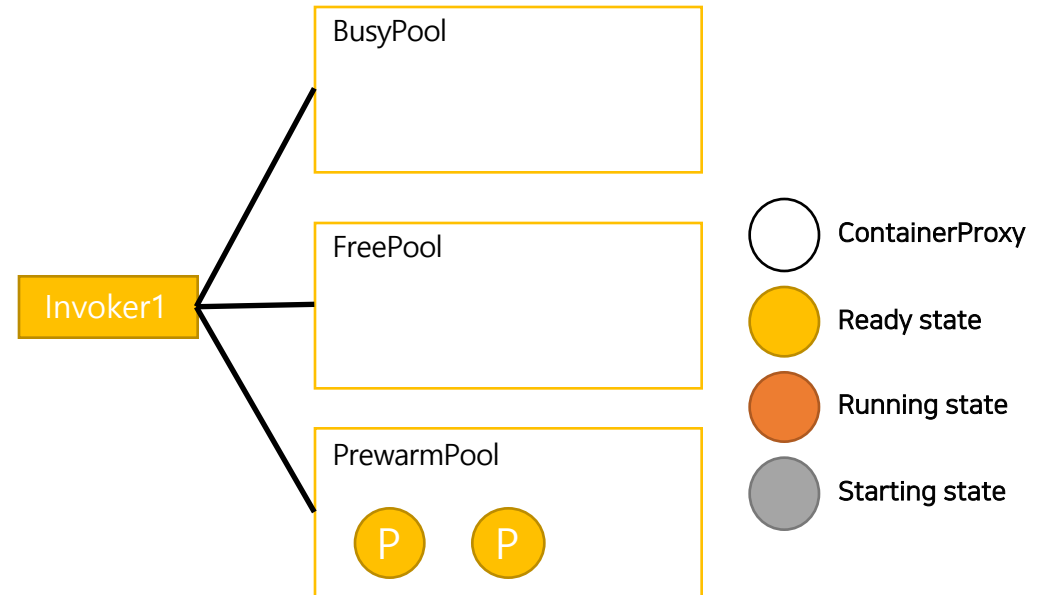
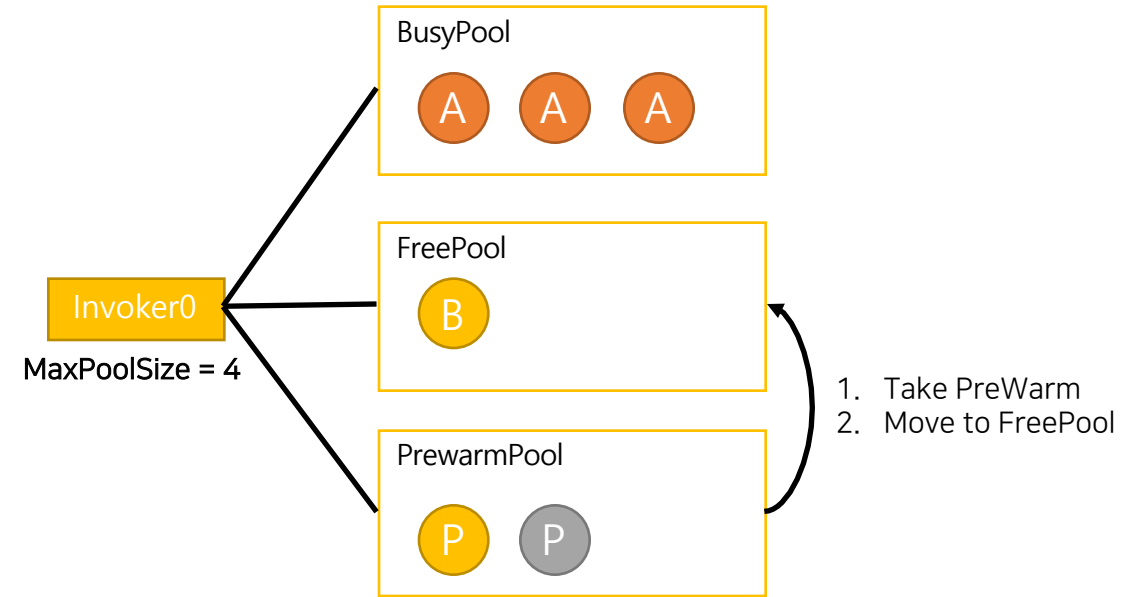
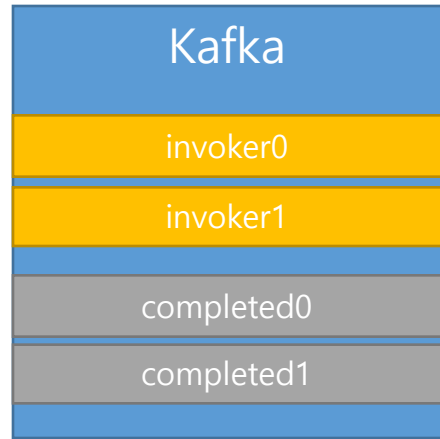
Container Deletion Flow: What happens in the real scene



Container Deletion Flow: What happens in the real scene

Controller0	
invoker0	0
invoker1	2

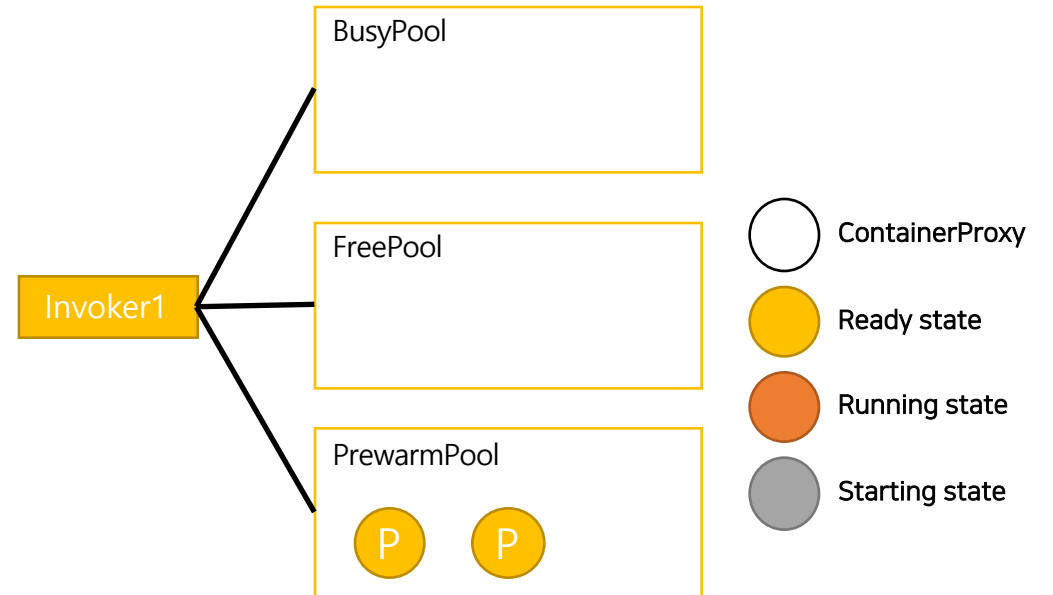
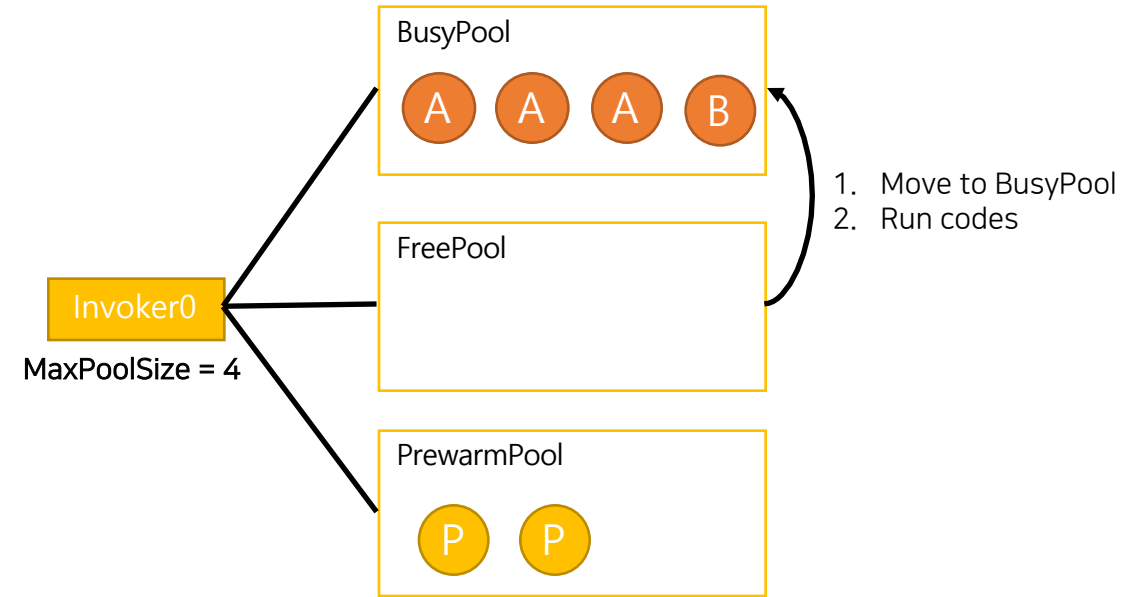
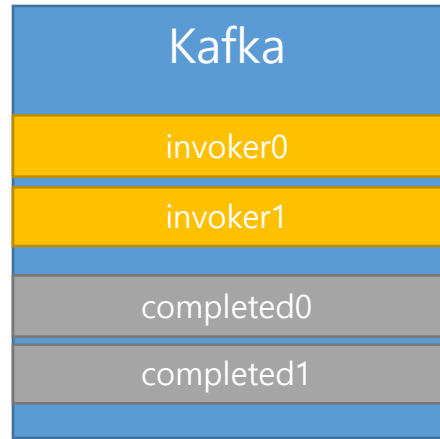
Controller1	
invoker0	0
invoker1	2



Container Deletion Flow: What happens in the real scene

Controller0	
invoker0	0
invoker1	2

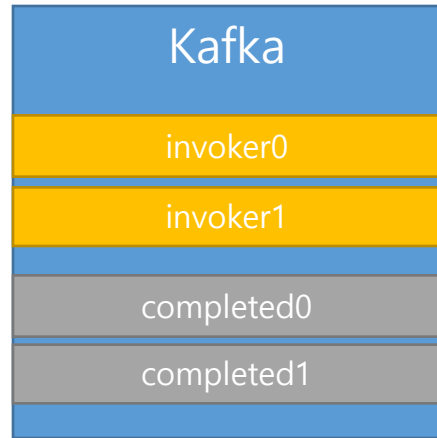
Controller1	
invoker0	0
invoker1	2



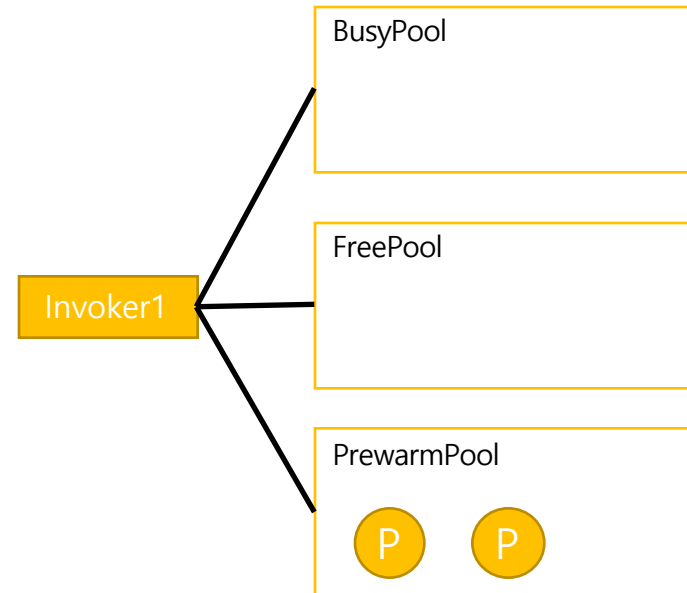
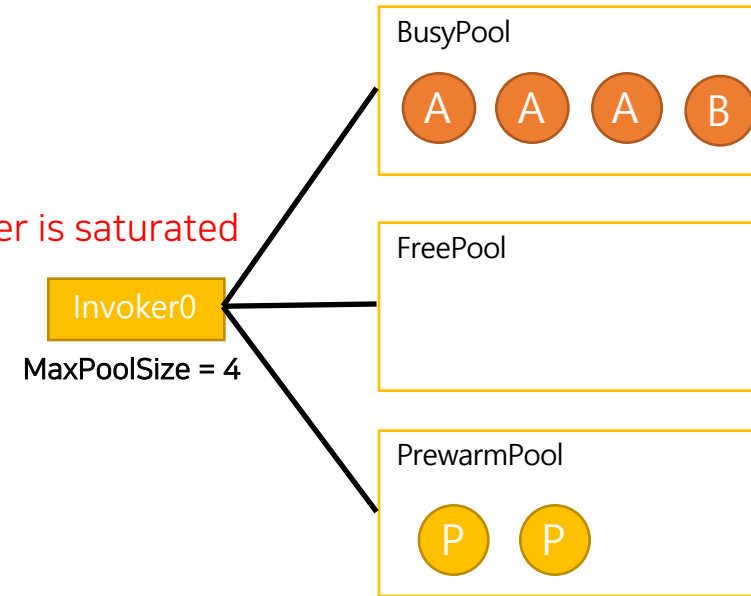
Choose Other than HomeInvoker: What happens in the real scene

Controller0	
invoker0	0
invoker1	2

Controller1	
invoker0	0
invoker1	2

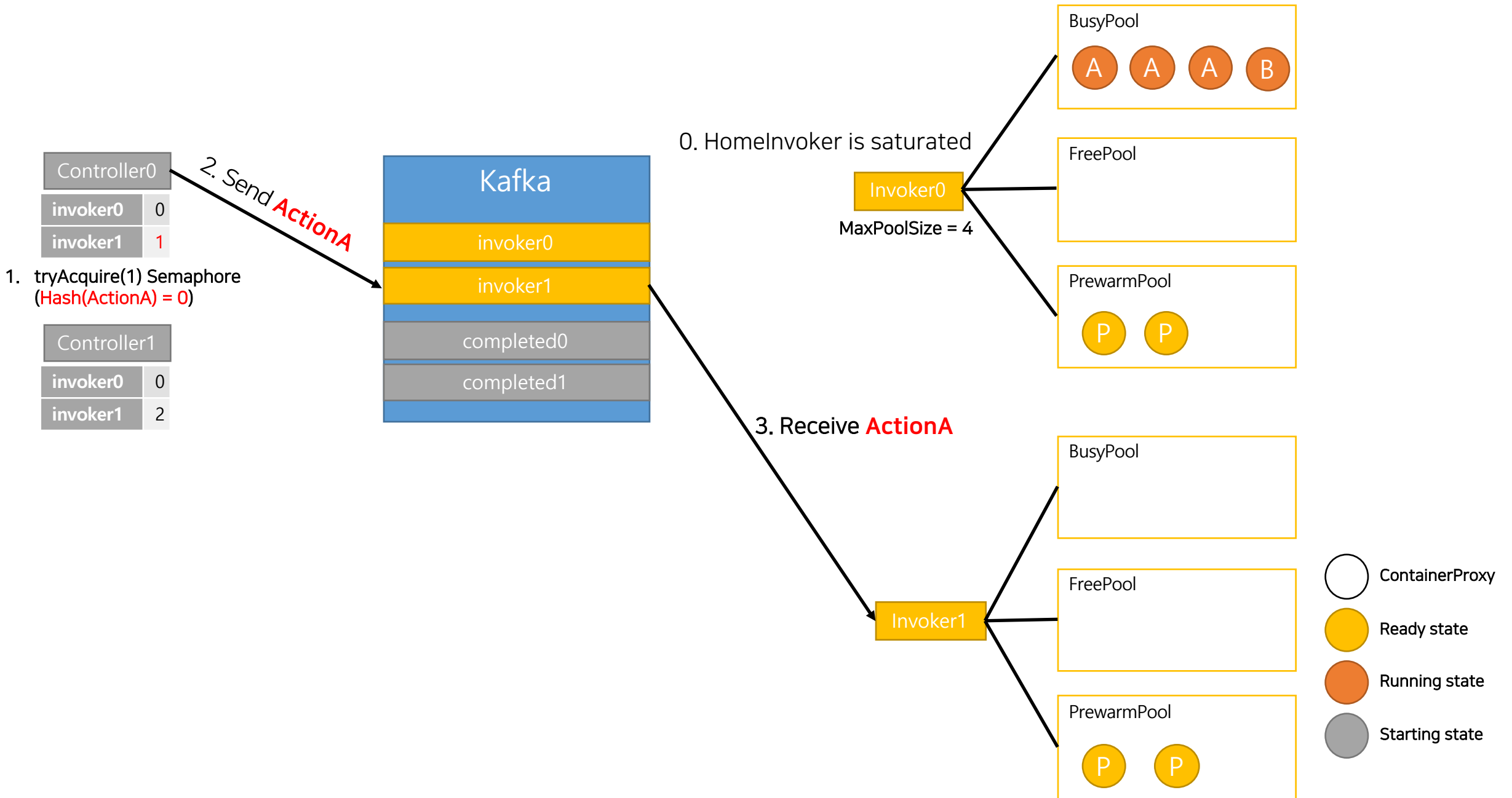


0. HomeInvoker is saturated

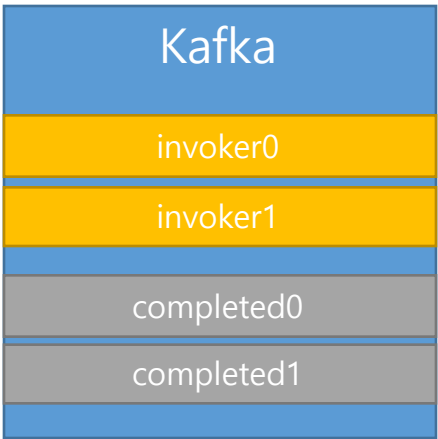
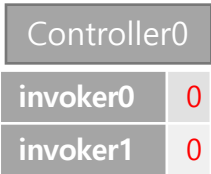


- ContainerProxy
- Ready state
- Running state
- Starting state

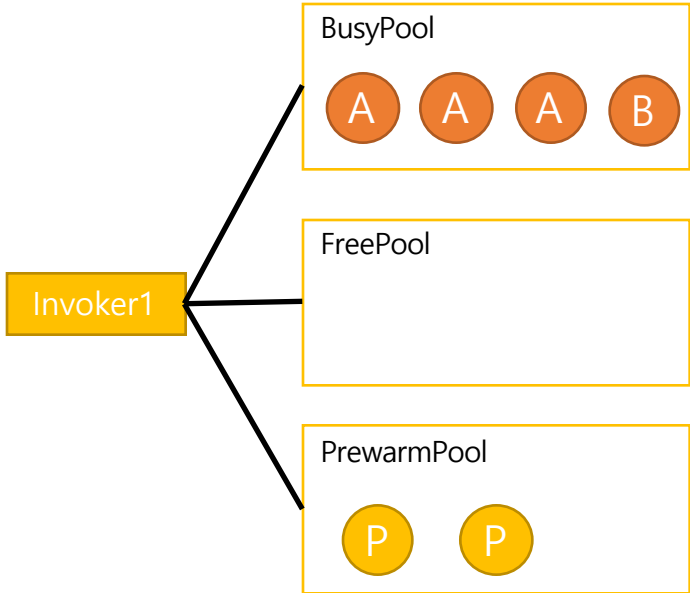
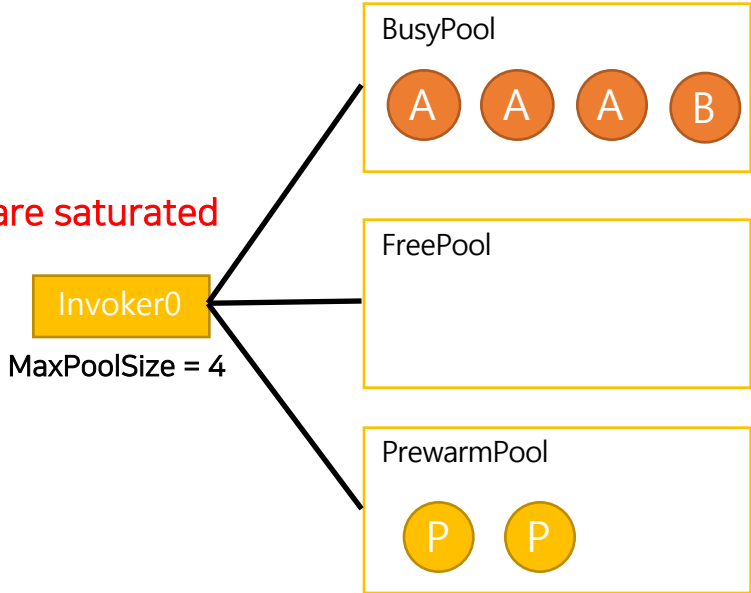
Choose Other than HomeInvoker: What happens in the real scene



Rescheduling Flow: What happens in the real scene

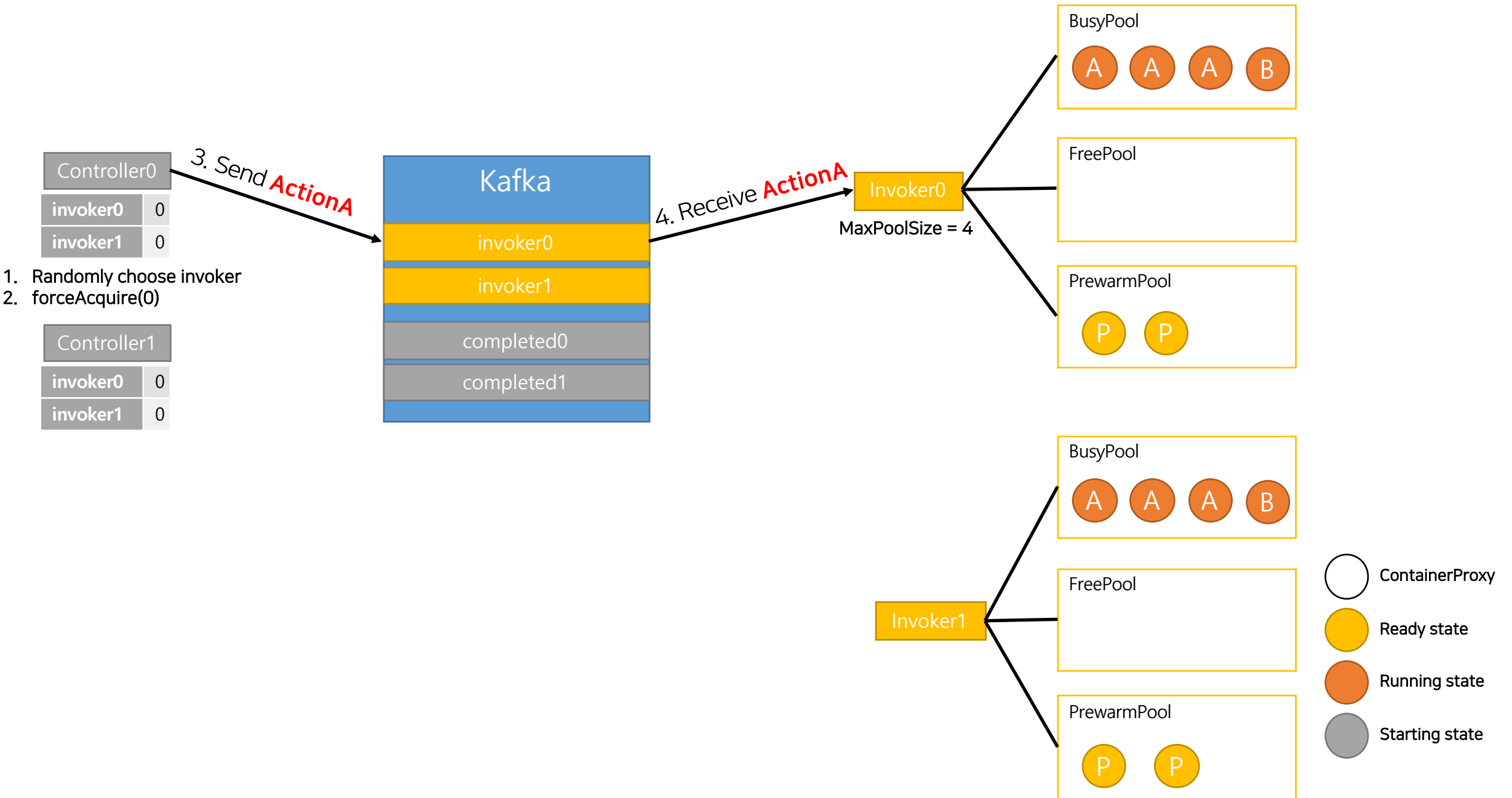


0. All invokers are saturated



- ContainerProxy
- Ready state
- Running state
- Starting state

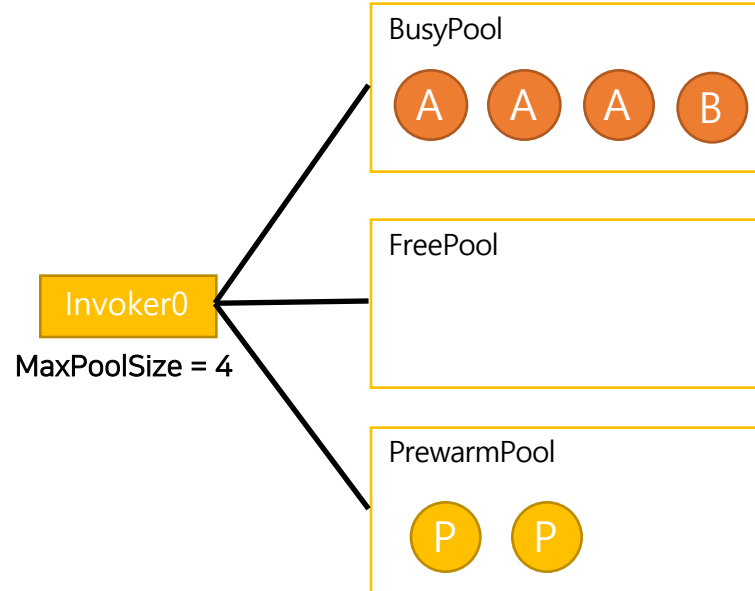
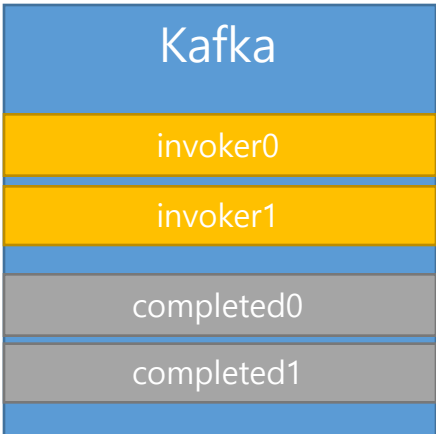
Rescheduling Flow: What happens in the real scene



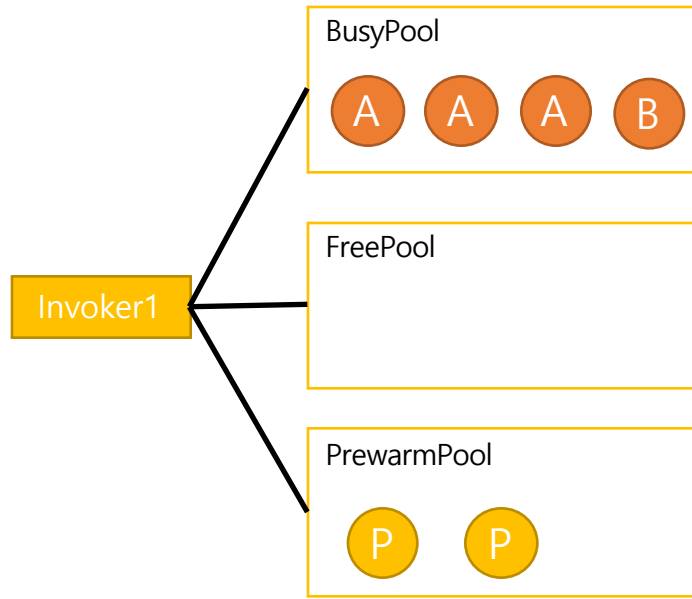
Rescheduling Flow: What happens in the real scene

Controller0	
invoker0	0
invoker1	0

Controller1	
invoker0	0
invoker1	0



1. Busy+Free = 4
2. Try removing from FreePool
3. Failed to acquire a container

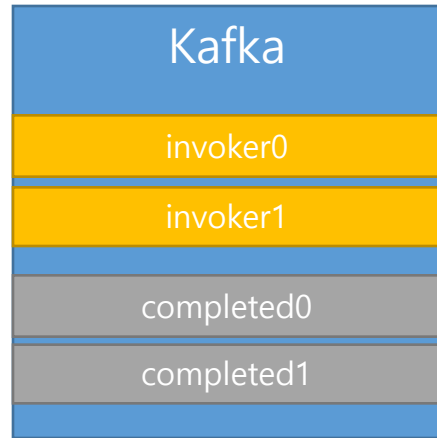


- ContainerProxy
- Ready state
- Running state
- Starting state

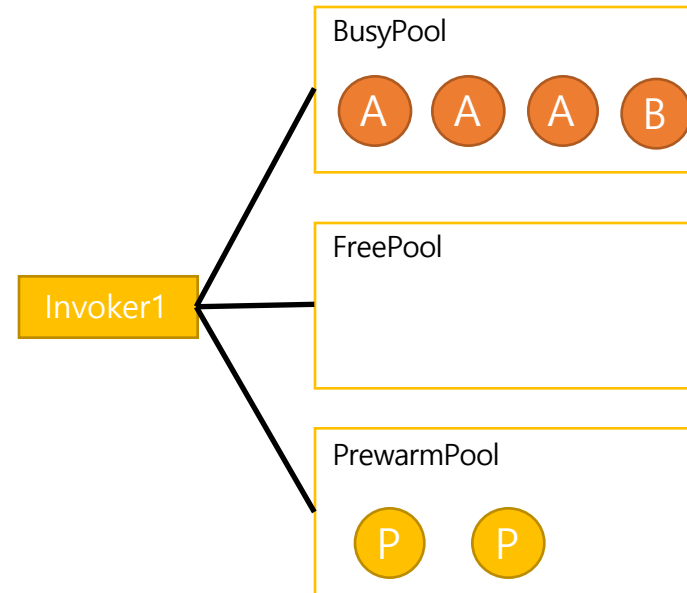
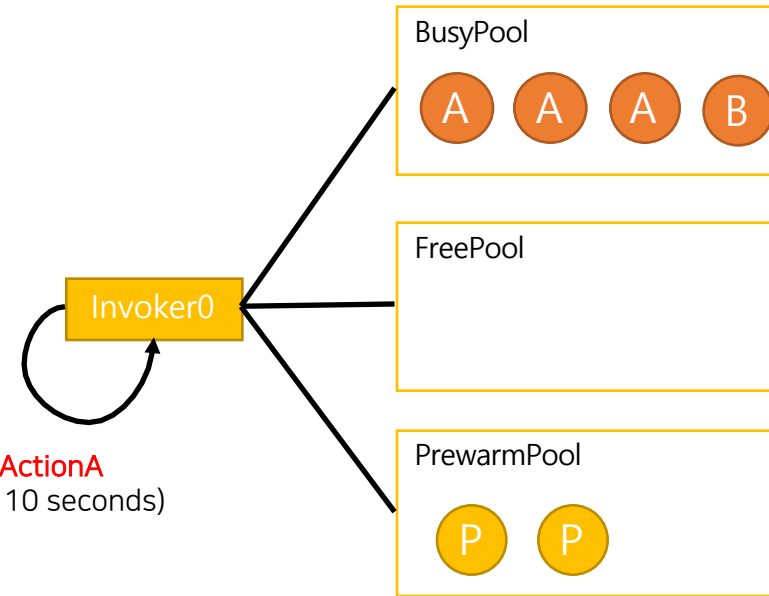
Rescheduling Flow: What happens in the real scene

Controller0	
invoker0	0
invoker1	0

Controller1	
invoker0	0
invoker1	0



1. Reschedule ActionA
(for at most 10 seconds)






Potential issues in current implementation

Issues in current implementation

- Hash is used to choose *HomeInvoker*.
- When scheduling invoker, *capacity of other invoker* is not considered.
- *Slow Docker command* is not considered.
 - Docker *pause/unpause* takes about *130~425ms*.
 - Docker *remove/create* takes about *700~1300ms*.

Problems in real scene – worst case: intervention of actions

-  Running container
-  Warmed container
-  Container is still being created

Invoker0 is saturated

ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

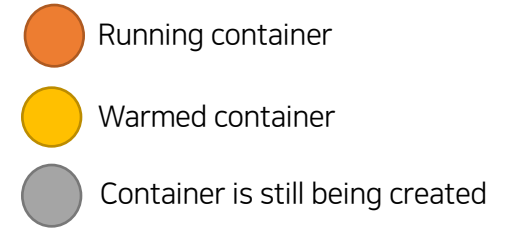
Invoker0
(slots: 4)

Invoker1
(slots: 4)

Invoker2
(slots: 4)



Problems in real scene – worst case: intervention of actions



0. **ExecutionA** finished

ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

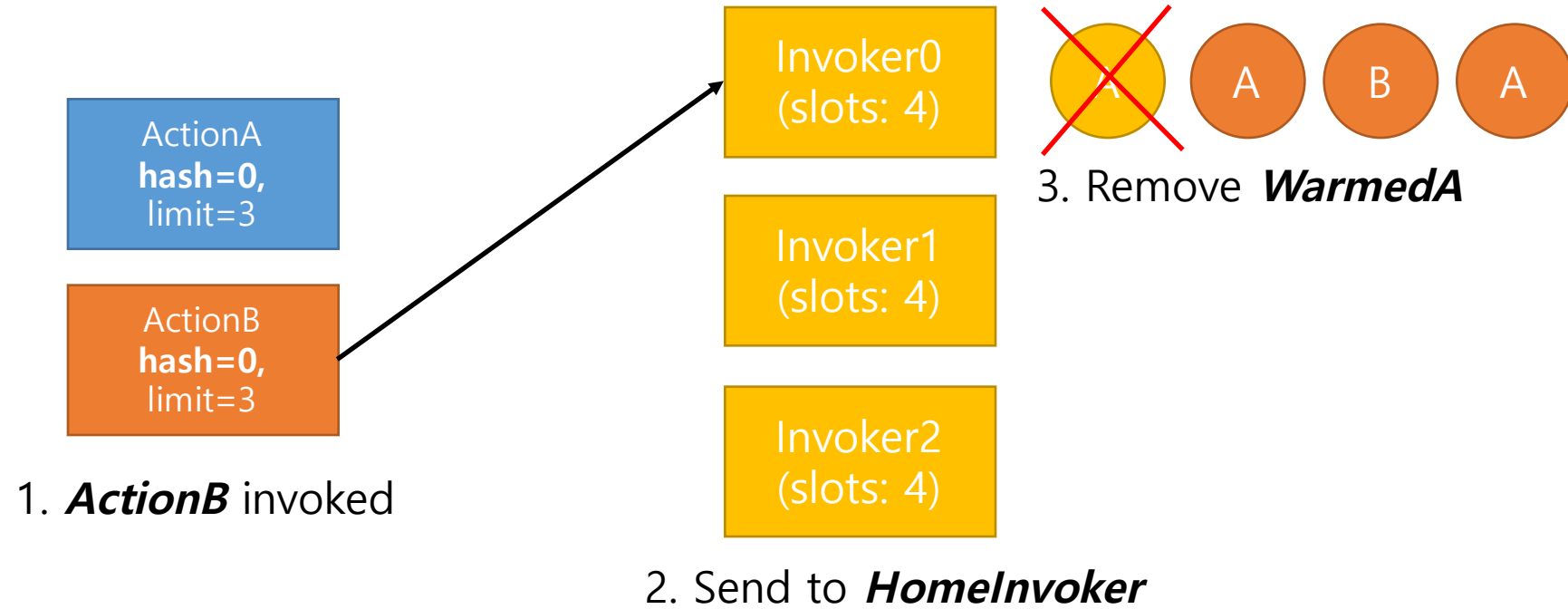
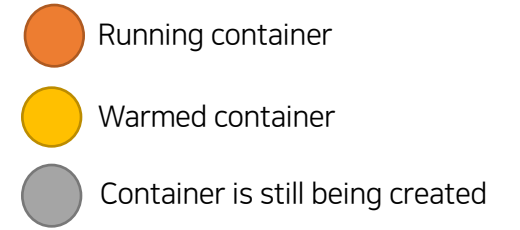
Invoker0
(slots: 4)

Invoker1
(slots: 4)




Invoker2
(slots: 4)



Problems in real scene – worst case: intervention of actions



Problems in real scene – worst case: intervention of actions

-  Running container
-  Warmed container
-  Container is still being created

ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

ActionB is running

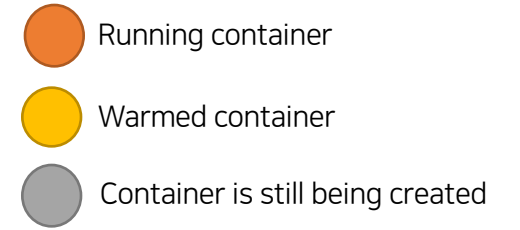
Invoker0
(slots: 4)

Invoker1
(slots: 4)

Invoker2
(slots: 4)



Problems in real scene – worst case: intervention of actions



ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

Invoker0
(slots: 4)

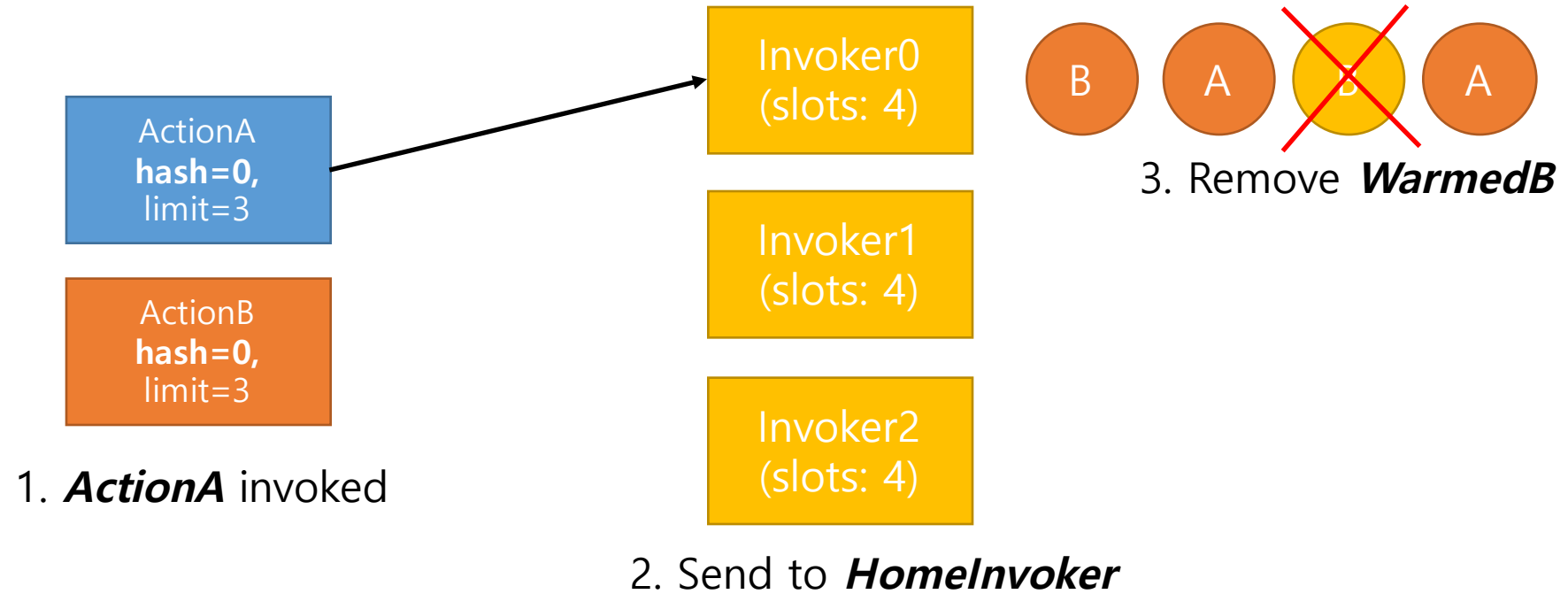
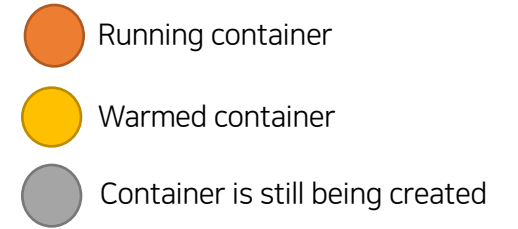
Invoker1
(slots: 4)

Invoker2
(slots: 4)




0. **ExecutionB** finished



Problems in real scene – worst case: intervention of actions



Problems in real scene – worst case: intervention of actions

-  Running container
-  Warmed container
-  Container is still being created

ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

Invoker0
(slots: 4)

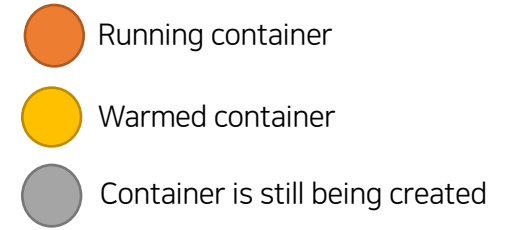
Invoker1
(slots: 4)

Invoker2
(slots: 4)

ActionA is running



Problems in real scene – worst case: intervention of actions



ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

Invoker0
(slots: 4)

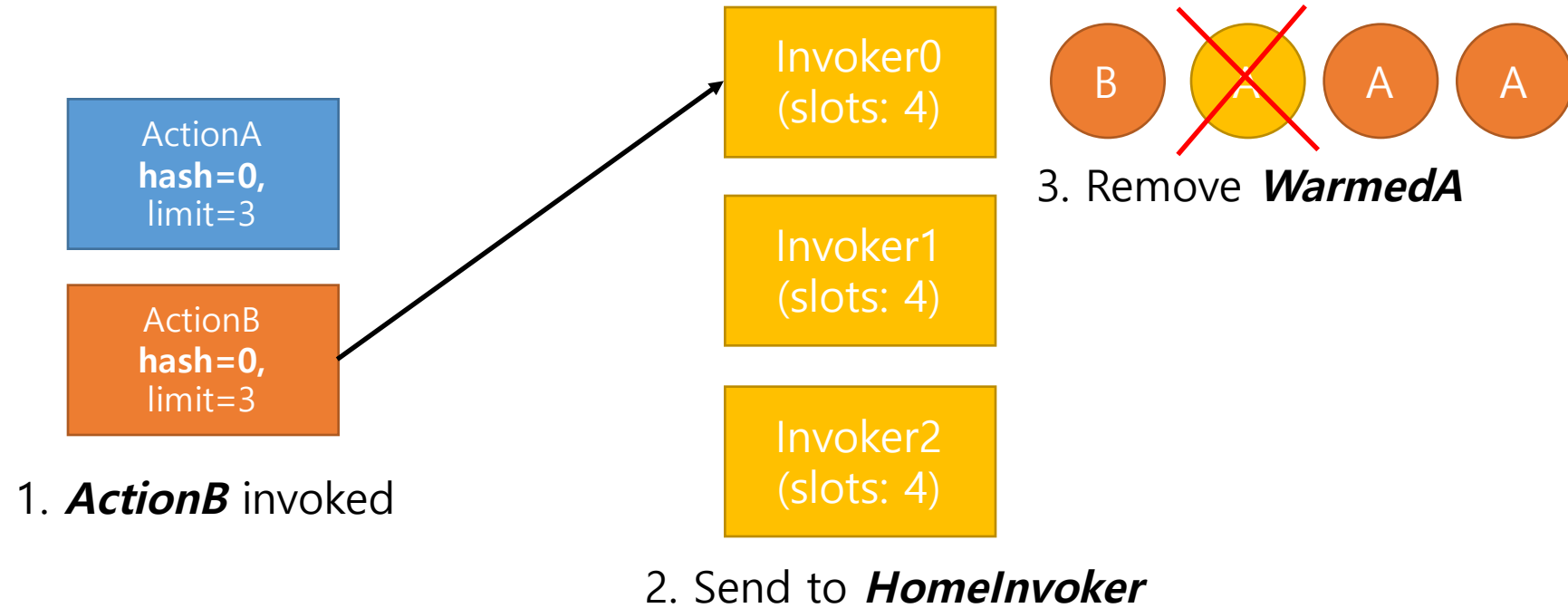
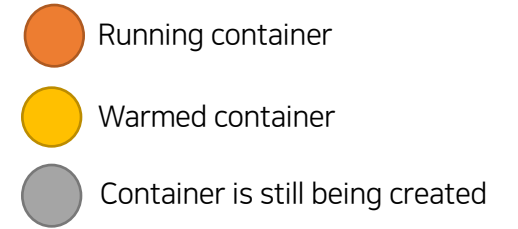
Invoker1
(slots: 4)

Invoker2
(slots: 4)

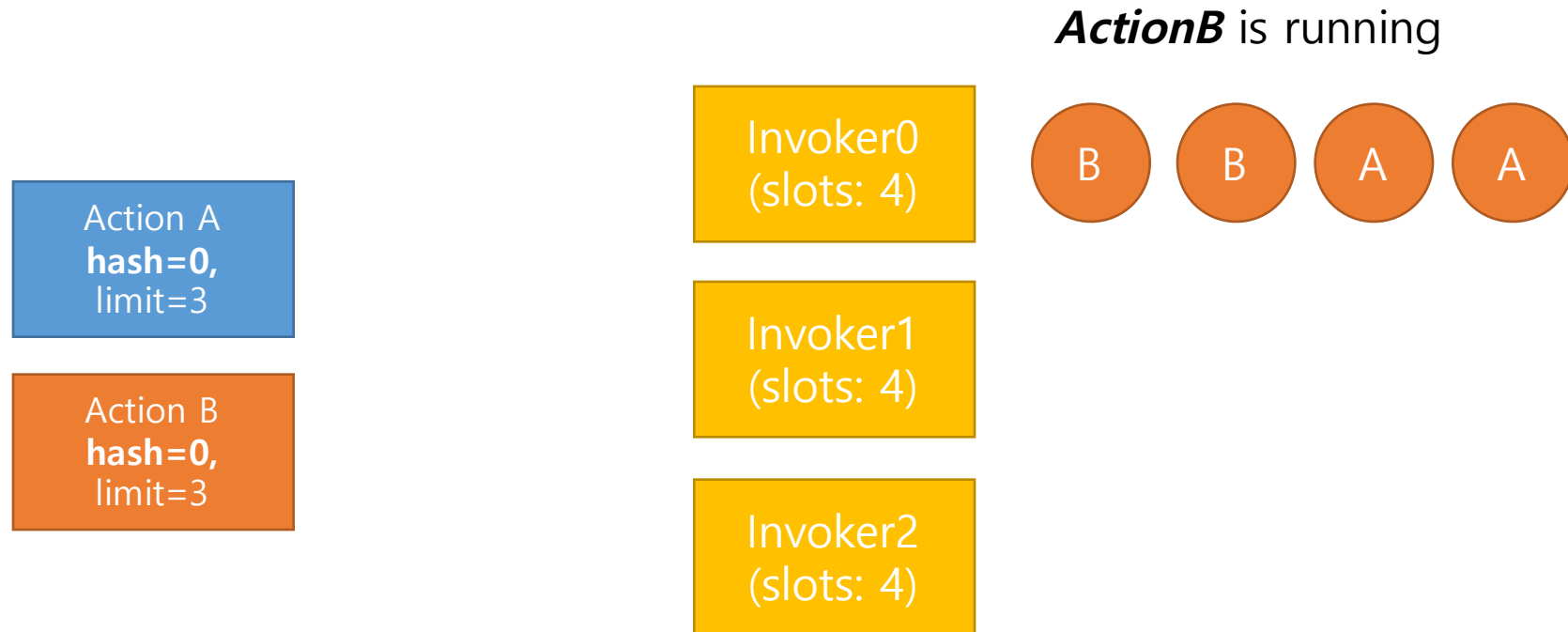
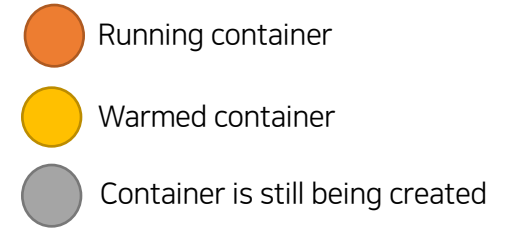
0. **ExecutionA** finished



Problems in real scene – worst case: intervention of actions

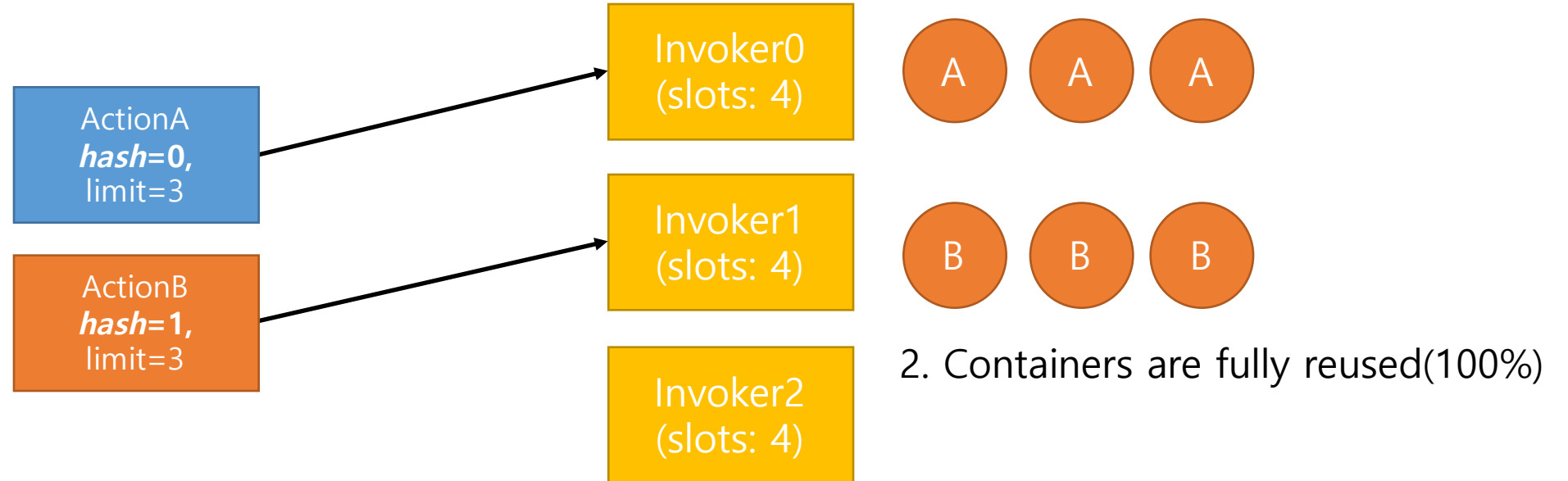
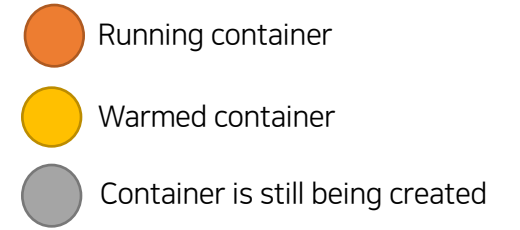


Problems in real scene – worst case: intervention of actions



Container Deletion(700ms) happens again and again because of same *HomeInvoker*

Problems in real scene – ideal case



1. Each actions is always sent to **HomeInvoker**

Problems in real scene – worst case2: invocation does not wait for completion

- Running container
- Warmed container
- Container is still being created
- P Prewarm container in starting state

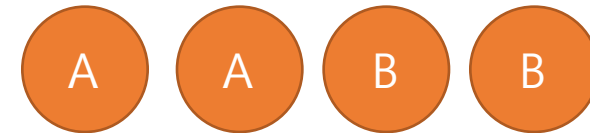
ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

Invoker0
(slots: 4)

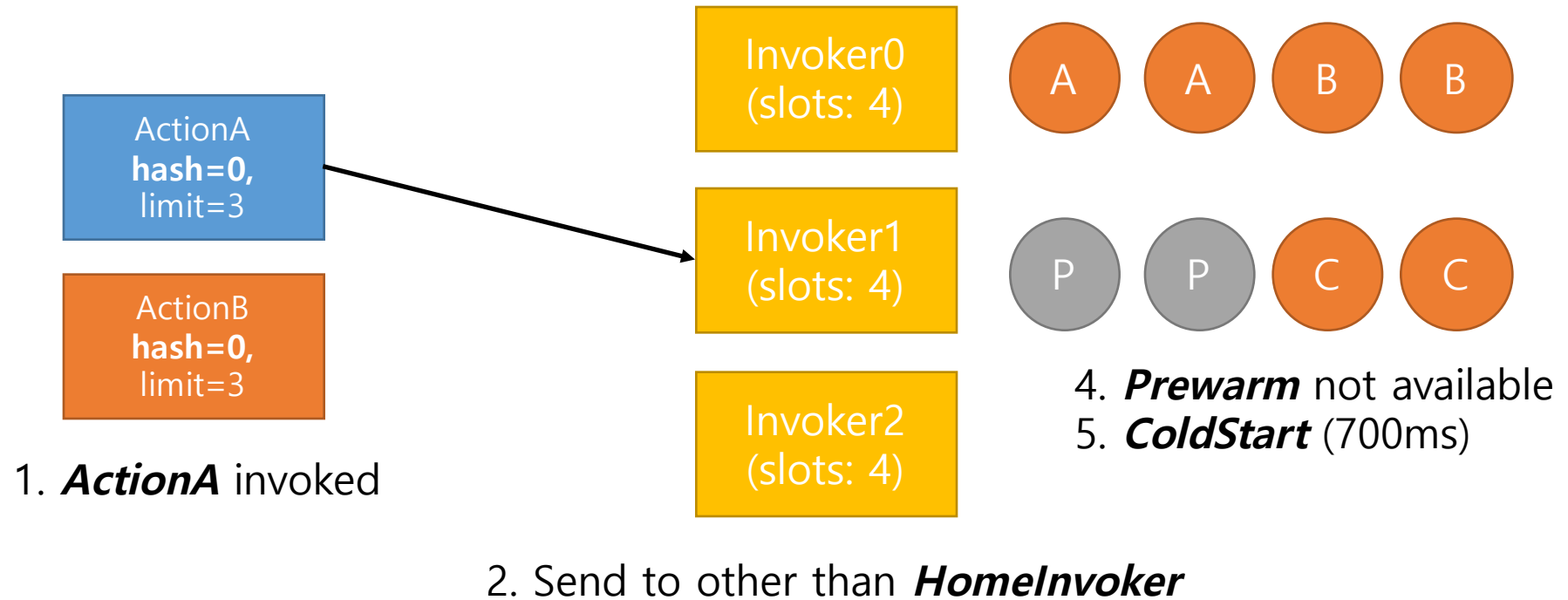
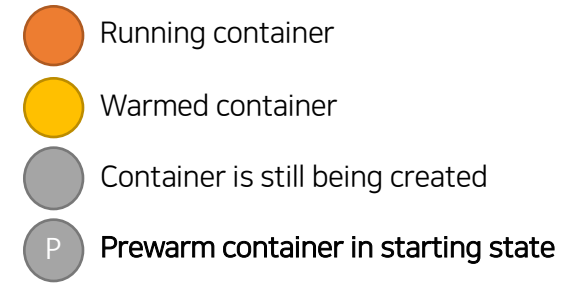
Invoker1
(slots: 4)

Invoker2
(slots: 4)

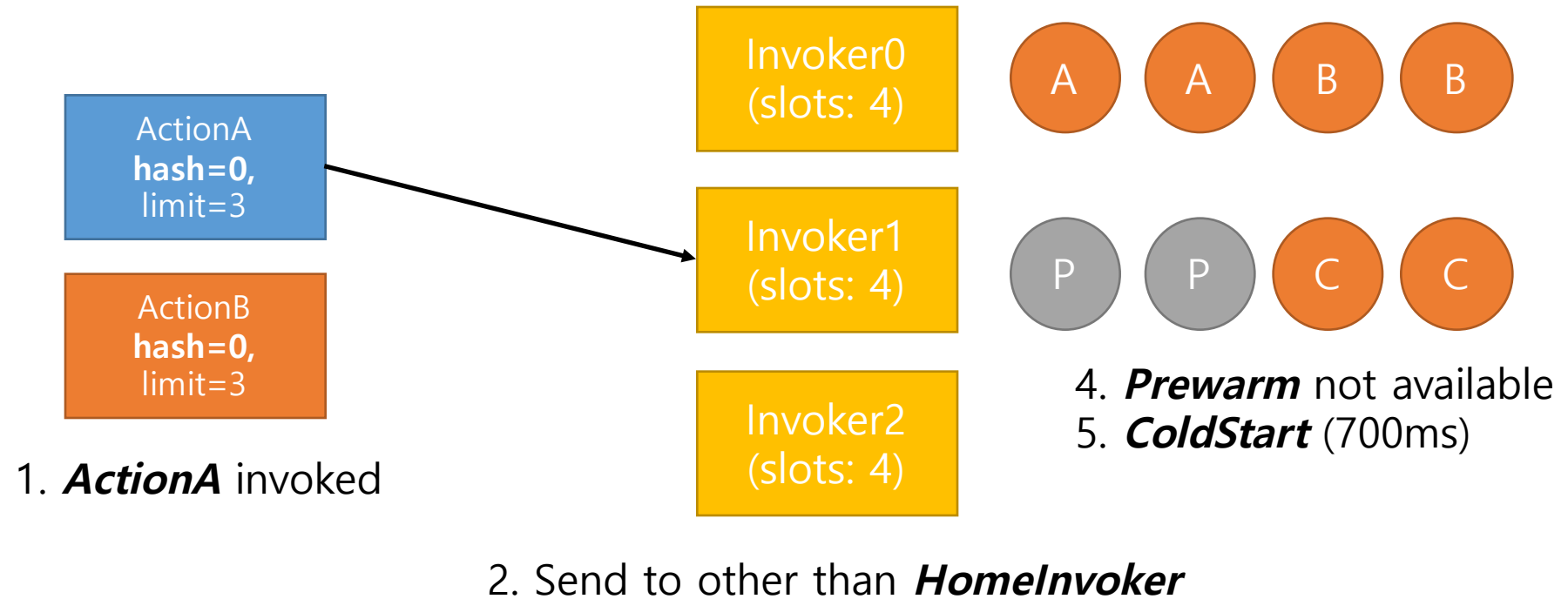


0. Previous requests already took all Prewarm containers

Problems in real scene – worst case2: invocation does not wait for completion

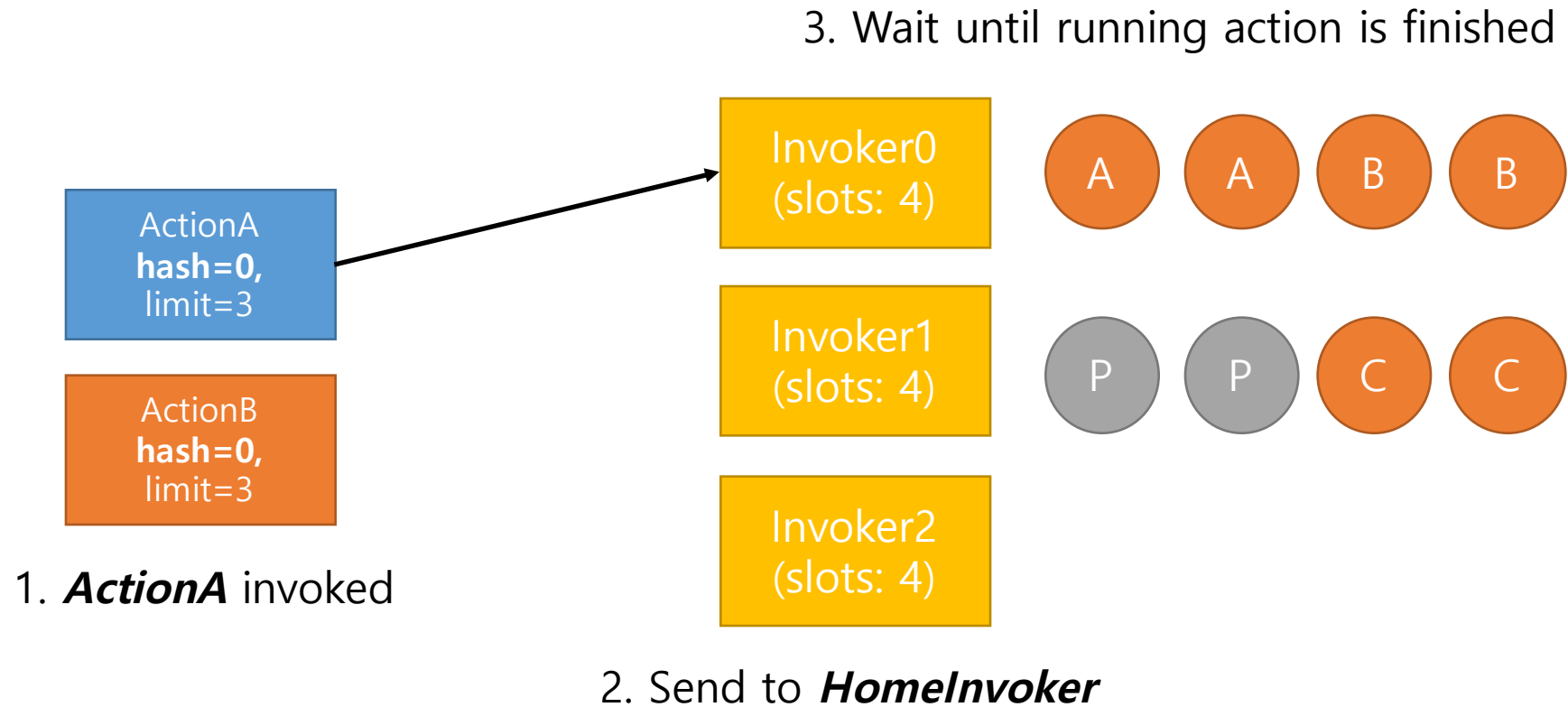


Problems in real scene – worst case2: invocation does not wait for completion



If execution time of **ActionA** is **20ms**, it takes **720ms** to run codes

Problems in real scene – ideal case: wait for completion

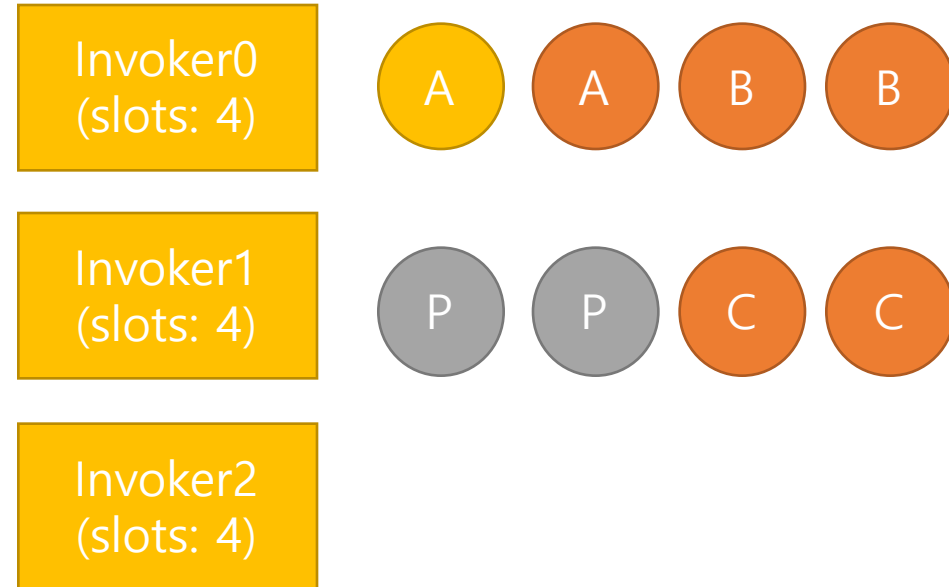


Problems in real scene – ideal case: wait for completion

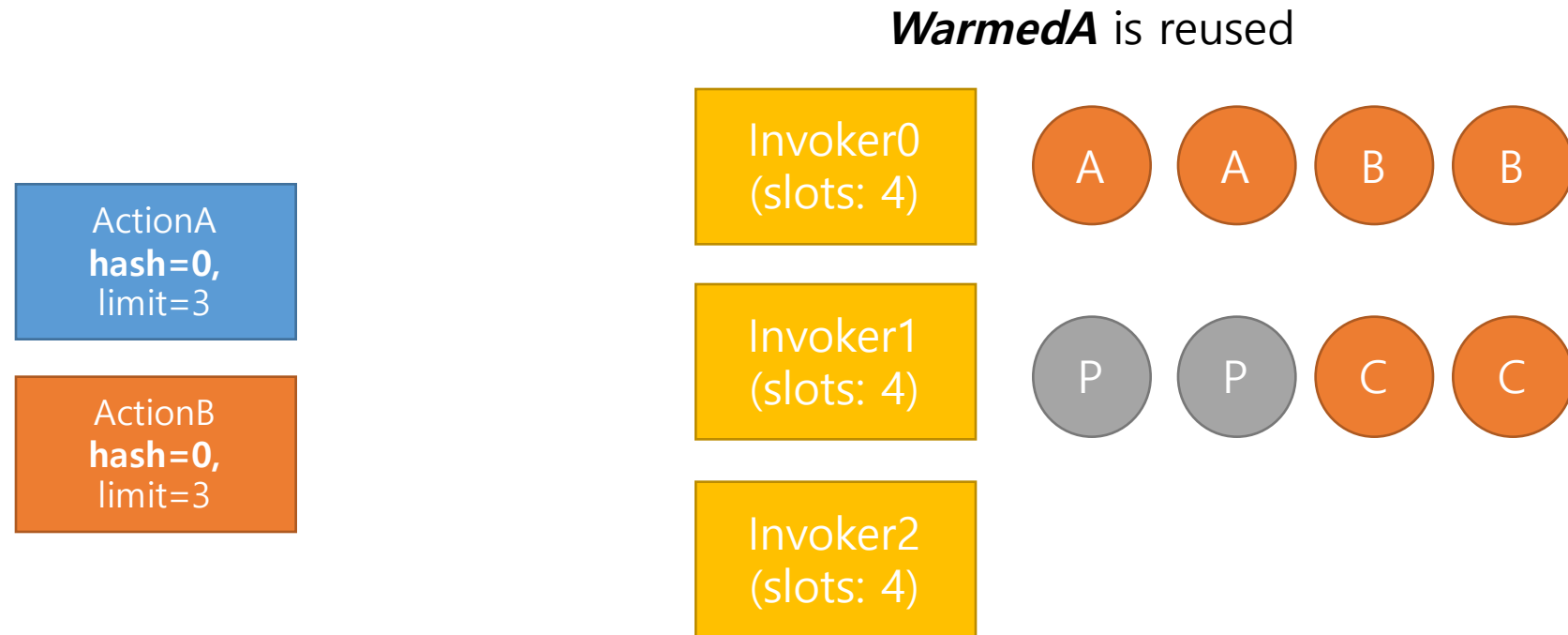
ActionA
hash=0,
limit=3

ActionB
hash=0,
limit=3

Reuse *WarmedA*

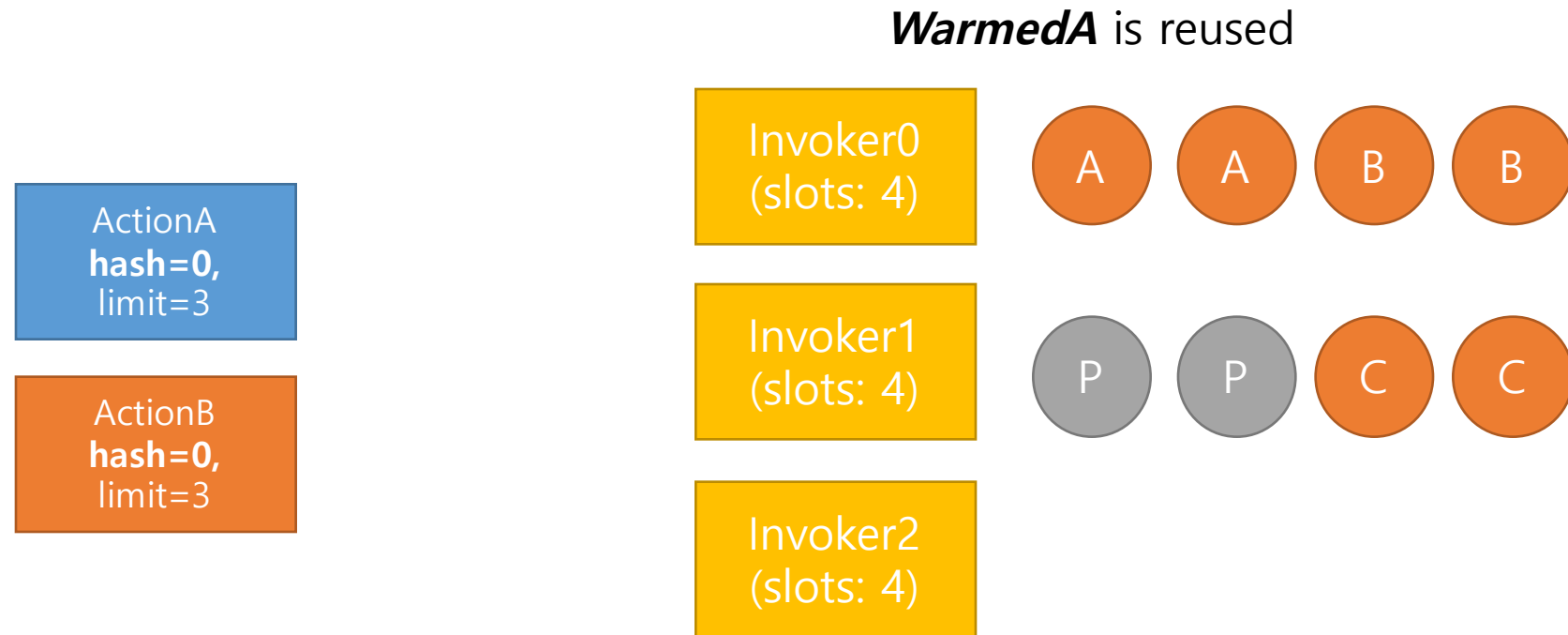


Problems in real scene – ideal case: wait for completion



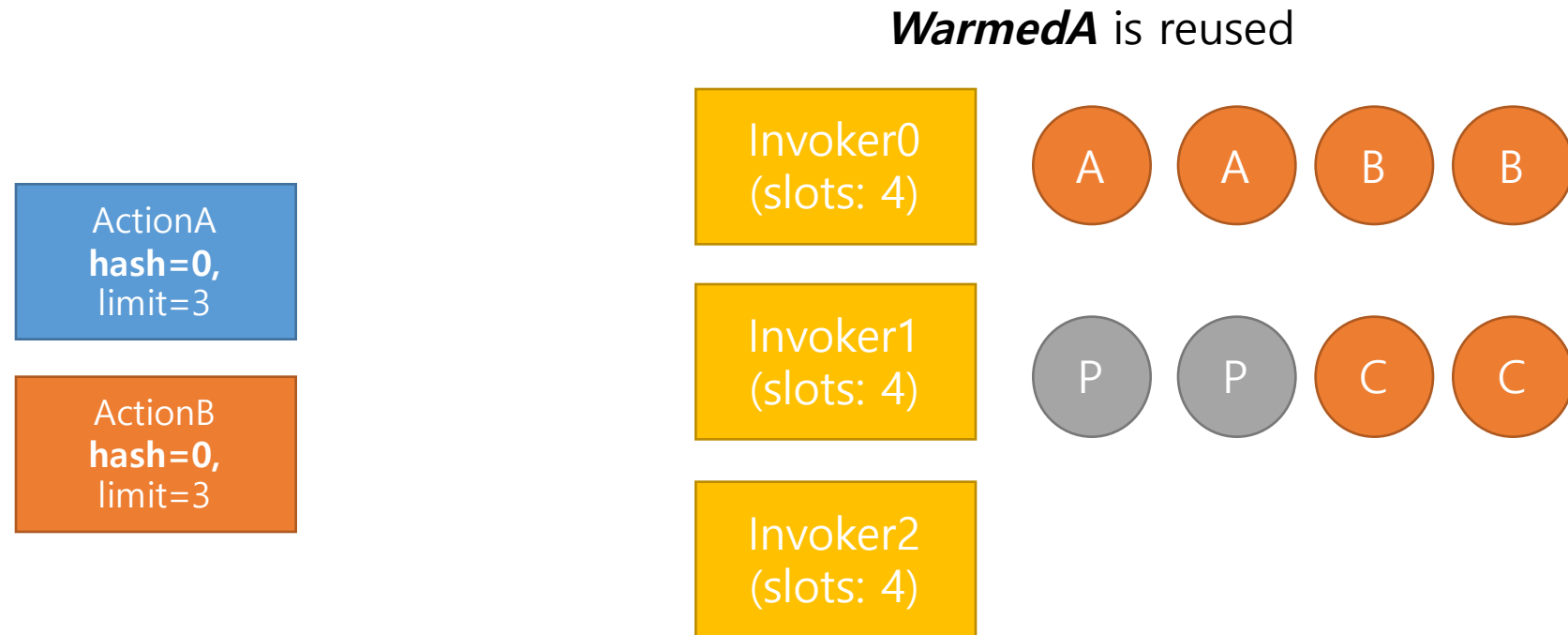
Wait for previous running(20ms) + Execute new run(20ms) = **40ms**

Problems in real scene – ideal case: wait for completion



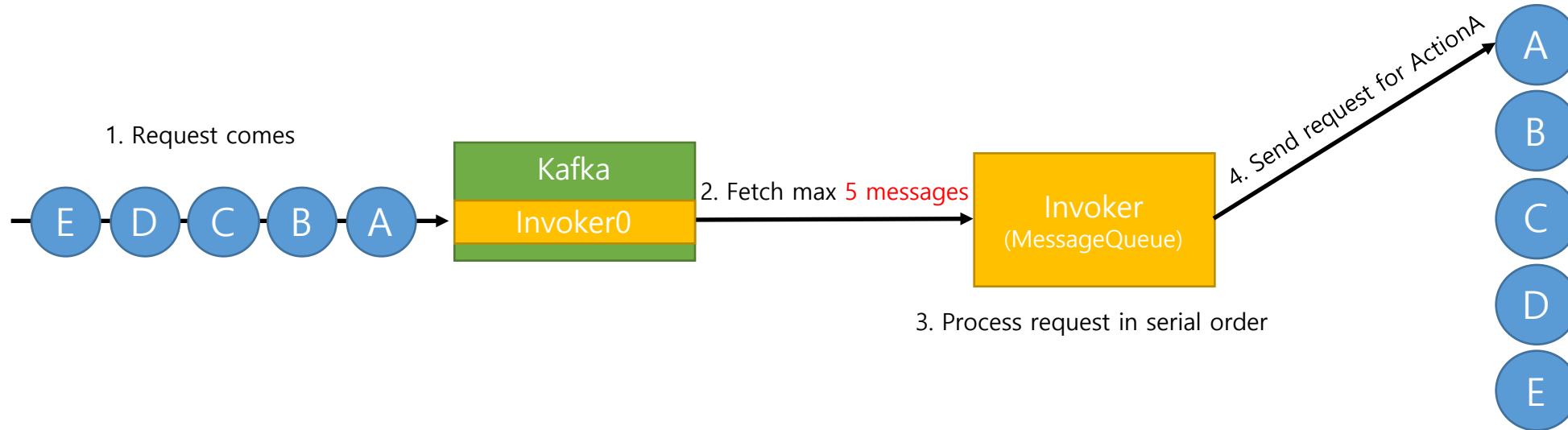
Wait for previous running(20ms) + Execute new run(20ms) = 40ms
720ms vs 40ms -> 18 times more

Problems in real scene – ideal case: wait for completion



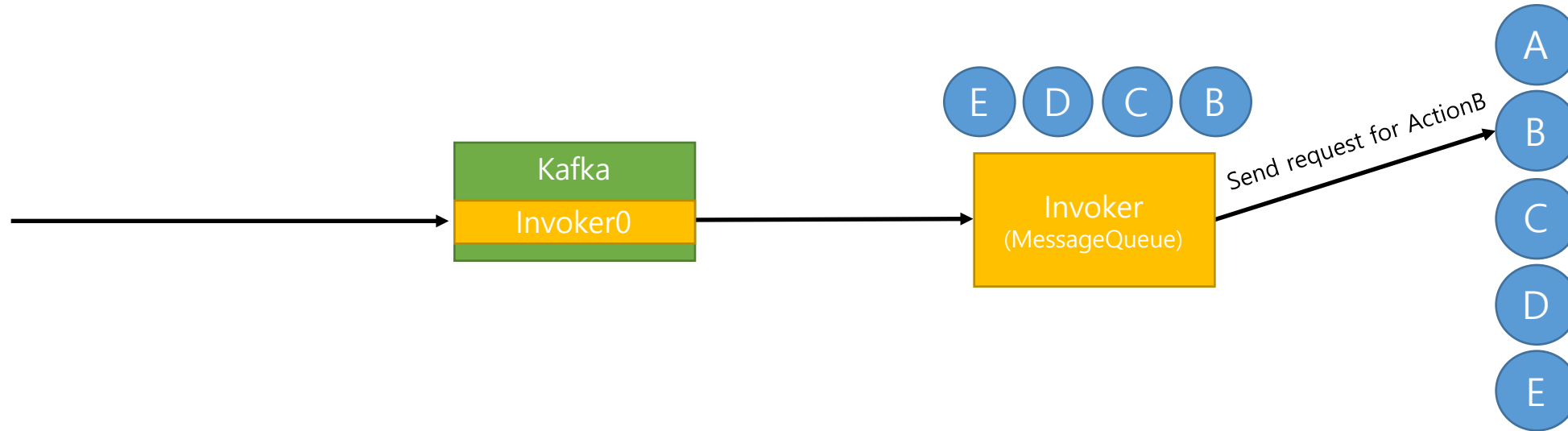
If *execution time* $\leq 700ms$, waiting is much faster than *ColdStart*.

Problems in real scene – Invoker coordination: blocking

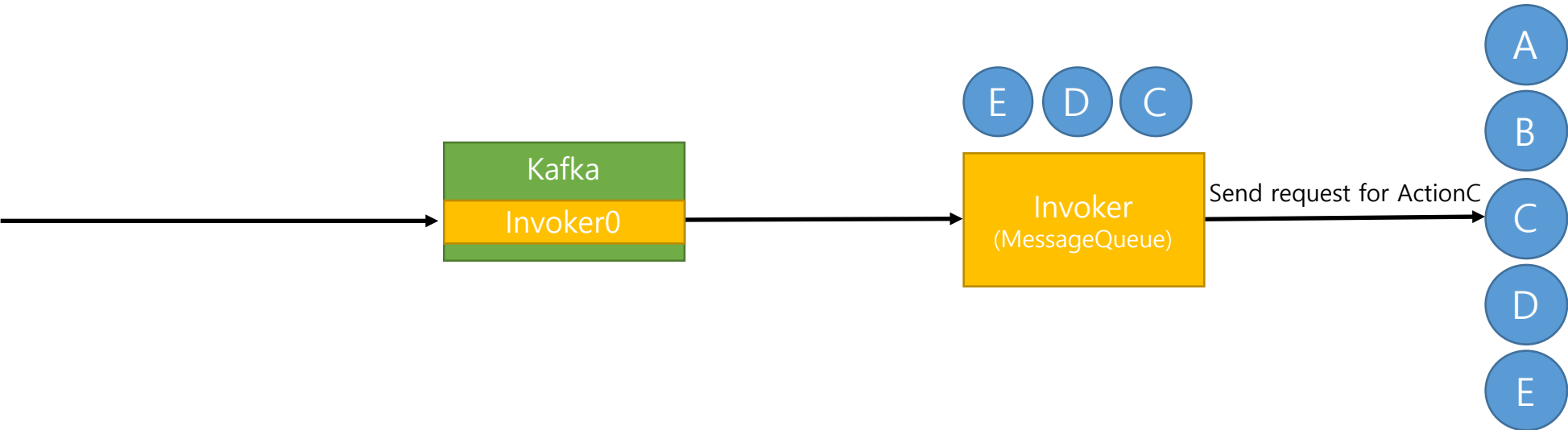


Invoker servers coordinate all messages in serial order.

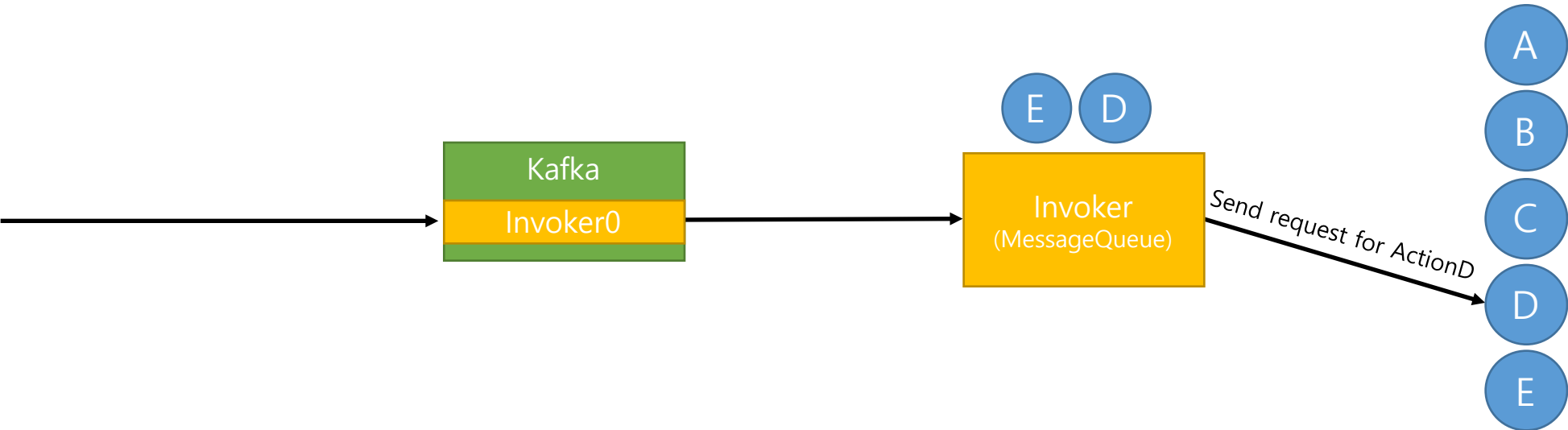
Problems in real scene – Invoker coordination: blocking



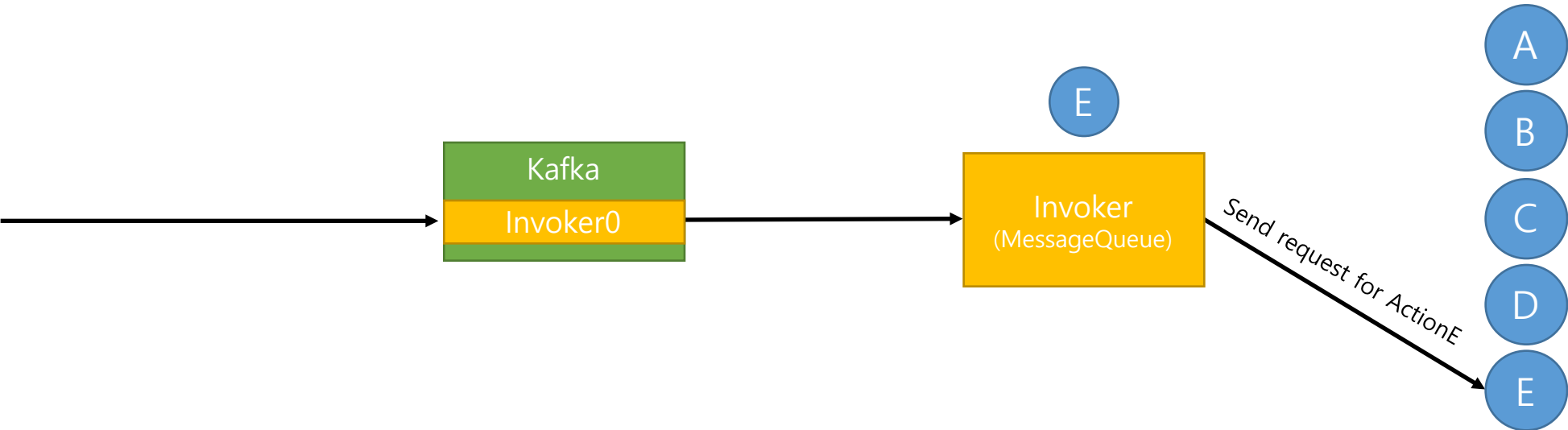
Problems in real scene – Invoker coordination: blocking



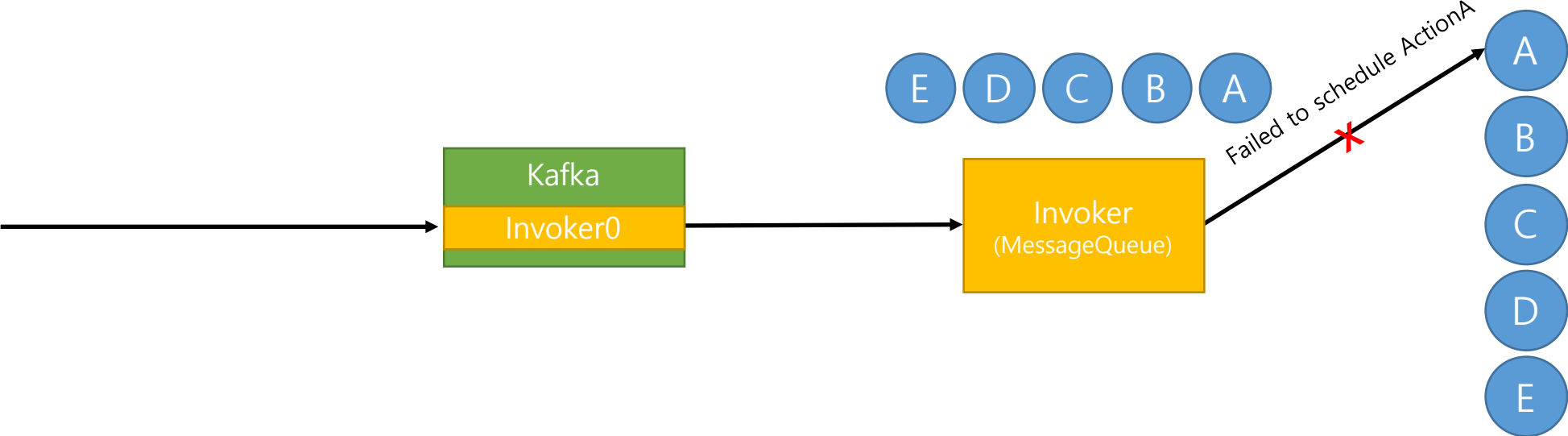
Problems in real scene – Invoker coordination: blocking



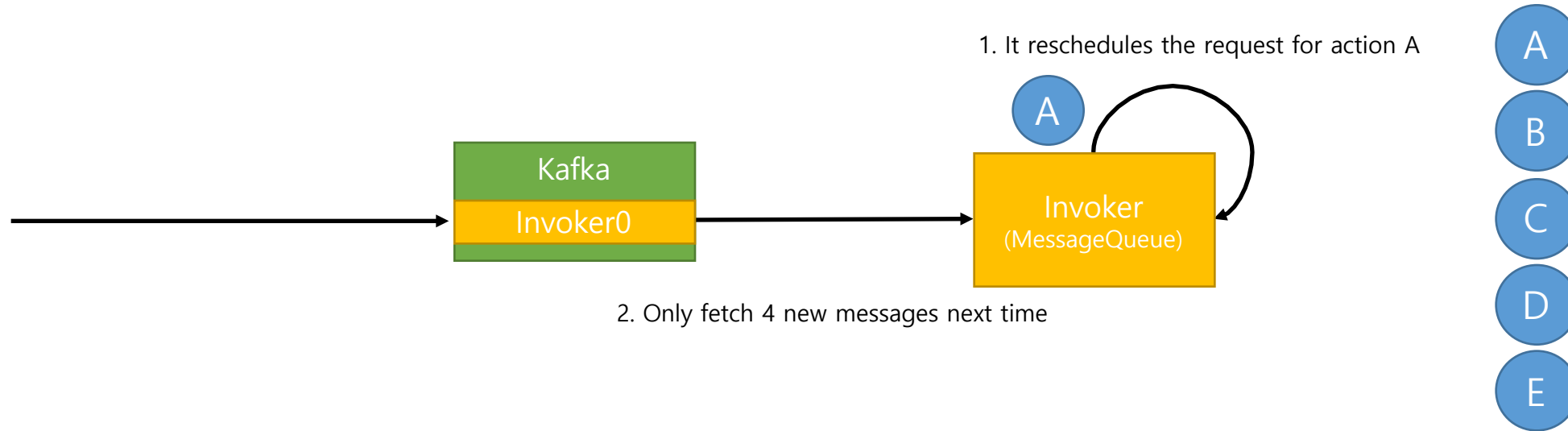
Problems in real scene – Invoker coordination: blocking



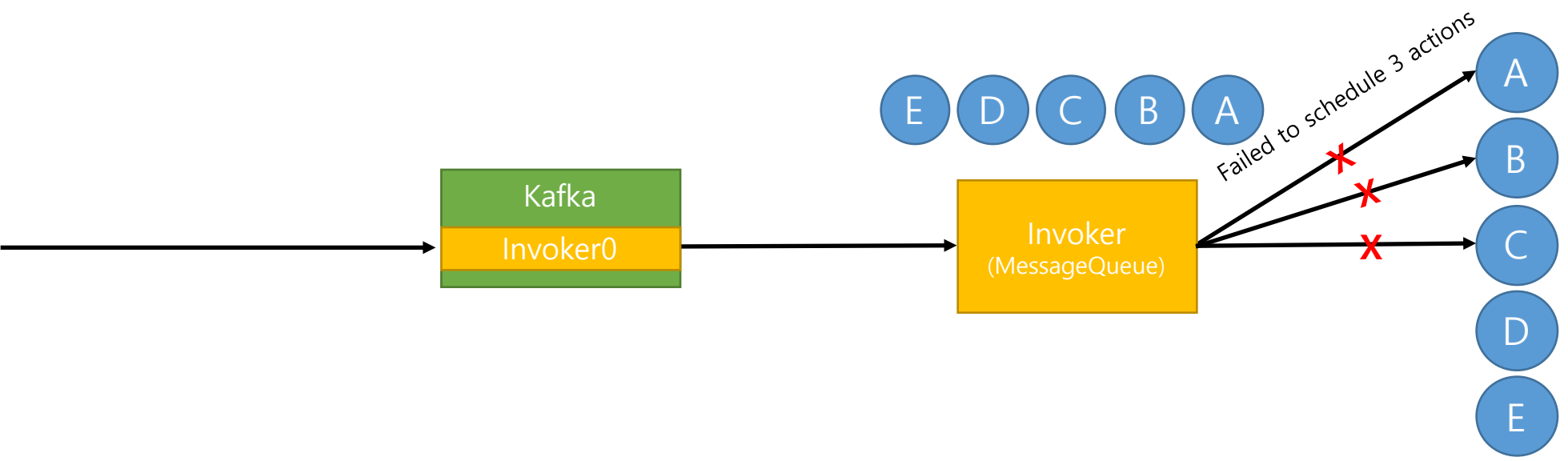
Problems in real scene – Invoker coordination: blocking



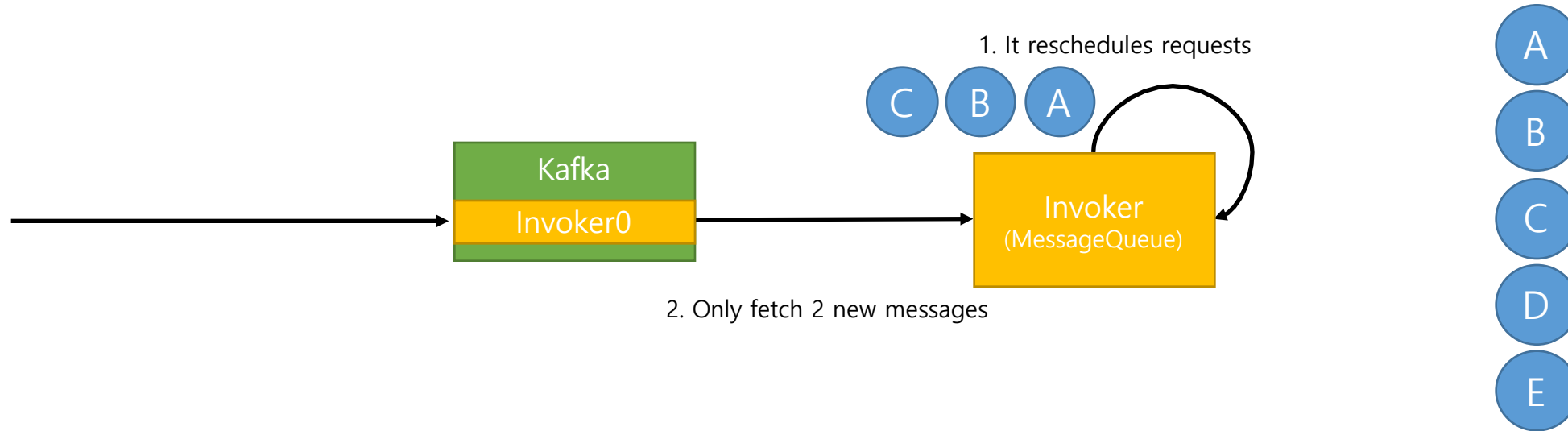
Problems in real scene – Invoker coordination: blocking



Problems in real scene – Invoker coordination: blocking

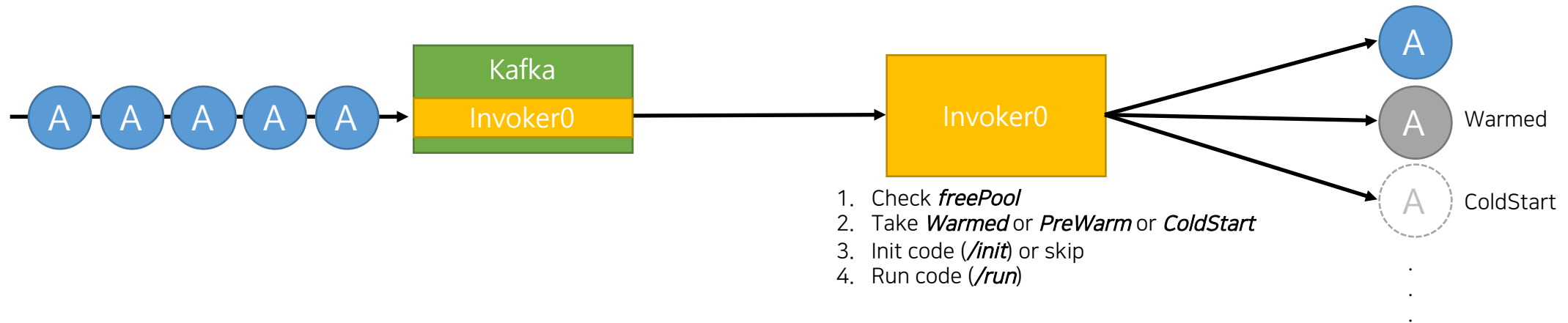


Problems in real scene – Invoker coordination: blocking



If some requests are blocked(rescheduled) for some reasons, it will affect invocation of subsequent requests.

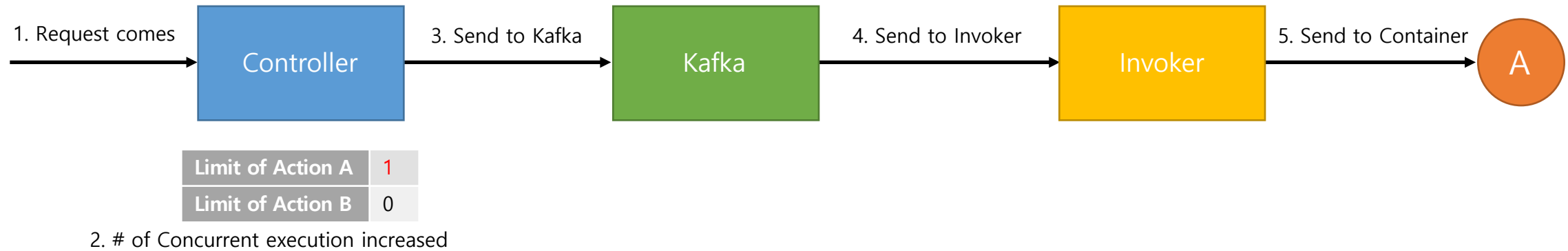
Problems in real scene – Invoker coordination: repeated logic overhead



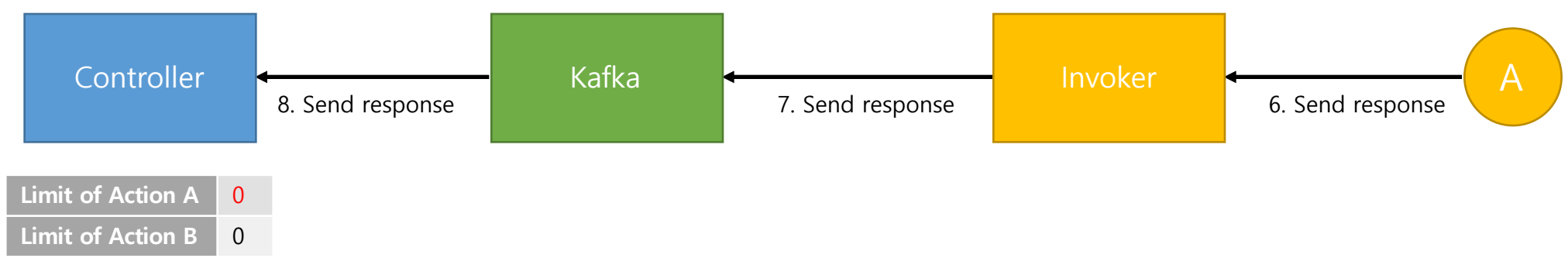
Even though there are jobs for only *ActionA*, invoker checks/proceeds all logics every time.

Problems in real scene - action concurrent execution limit

concurrentInvocations : 1
invocationsPerMinute : 1
firesPerMinute : 1

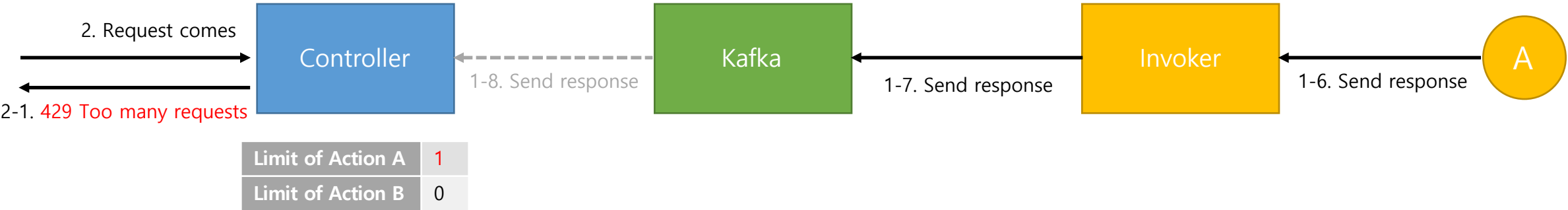


Problems in real scene - action concurrent execution limit

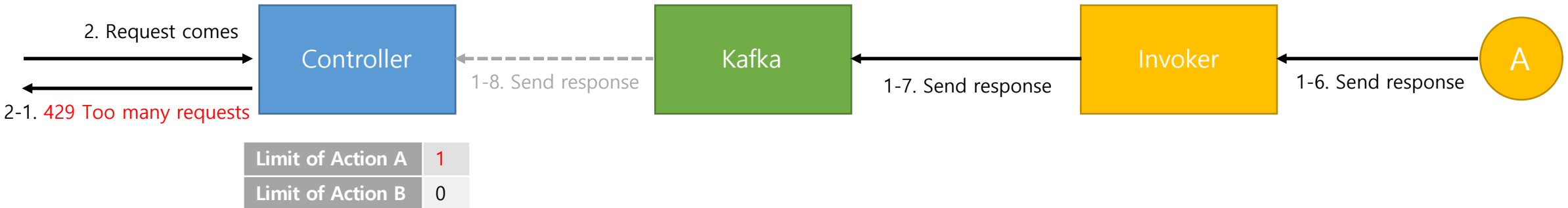


9. # of Concurrent execution decreased

Problems in real scene - action concurrent execution limit



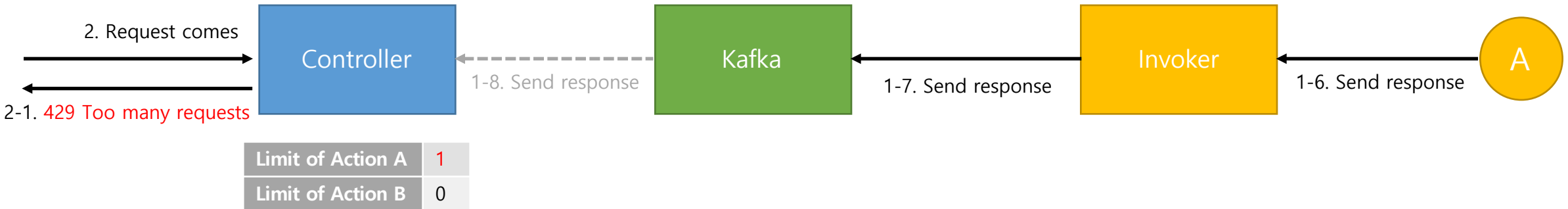
Problems in real scene - action concurrent execution limit



Even though *# of concurrent running containers* is lesser than *limit*, controller always returns **429 Too many requests** until it receives completion message.

(When I configured limit as 5, I could only invoke about 3 containers.)

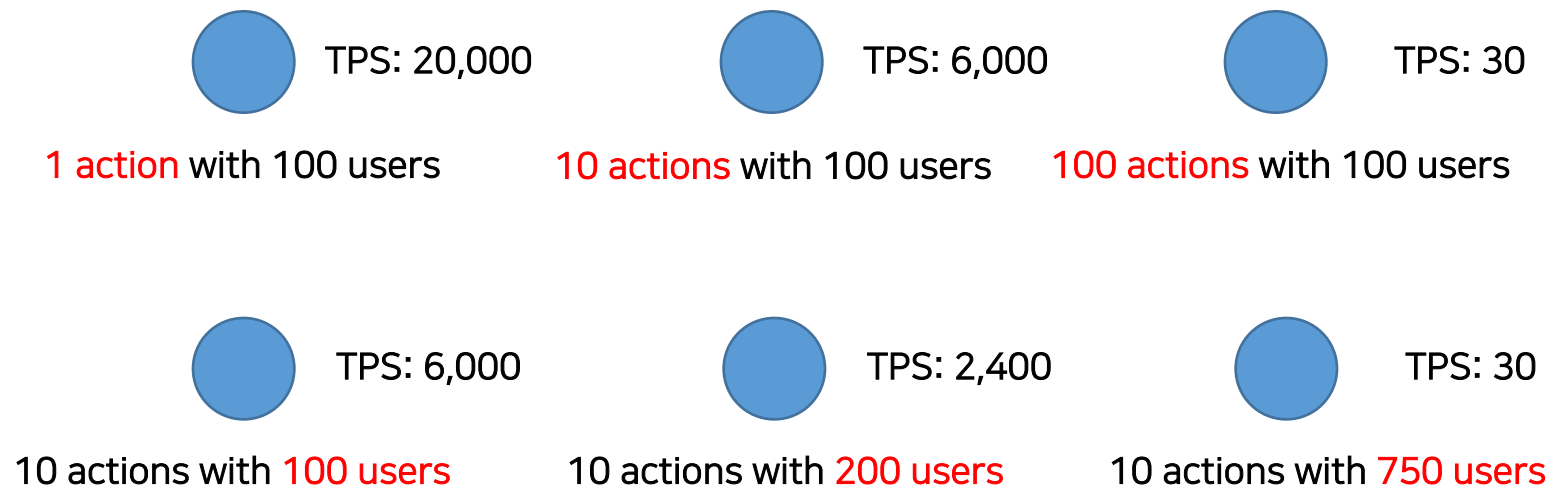
Problems in real scene - action concurrent execution limit



Even if it is intended behavior, if we can control resources based on real usage(container usage), it will allow more fine-grained throttling.

Problems in real scene – Nondeterministic performance

- In current architecture, it's not easy to determine **when do we have to add more servers.**
 - Though other invoker has enough capacity, TPS is less if *homeInvoker* of actions are same.
 - **TPS is not proportional to the number of invokers.** -> If intervention occurred, TPS decreases, not occurred, TPS increases.
 - It's not easy to advertise our official TPS.
- **TPS is highly dependent on container creation/deletion.**
 - If *coldStart* is occurred, execution time becomes at least 700ms.
- **TPS is changed as the number of users and the number of actions are changed.**



We cannot say our system supports either 20K TPS or 30 TPS.

Problems in real scene – Nondeterministic performance

- In current architecture, it's not easy to determine **when do we have to add more servers.**
 - Though other invoker has enough capacity, TPS is less if *homeInvoker* of actions are same.
 - **TPS is not proportional to the number of invokers.** -> If intervention occurred, TPS decreases, not occurred, TPS increases.
 - It's not easy to advertise our official TPS.
- **TPS is highly dependent on container creation/deletion.**
 - If *coldStart* is occurred, execution time becomes at least 700ms.
- TPS is changed as **the number of users and the number of actions** are changed.

Invoker *maxPoolSize* = 20

1 Container = 100 TPS
1 Invoker = 2,000 TPS
10 Invokers = 20,000 TPS

New user requirement
→
20,000 TPS

Add 10 more invokers

It would be great if we can calculate expected TPS arithmetically

Problem summary

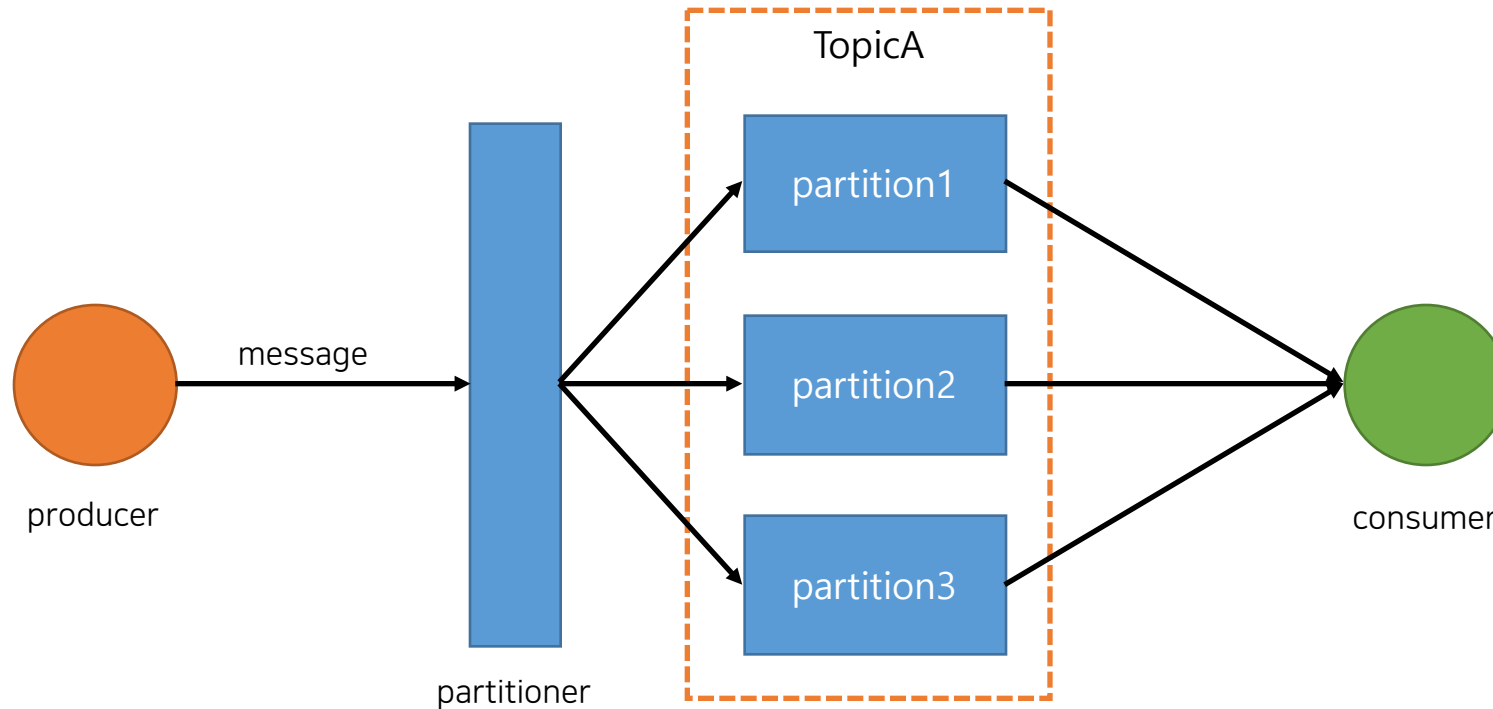
- *Intervention* of actions: Hash(no scheduling based on capacity and status)
- Invocation does not wait for previous run: Wait for *coldStart*
- Invoker coordinates all messages: *blocking, logic overhead*
- Action concurrent execution limit: *can have more fine-grained control over resources*
- TPS is highly dependent on container creation/deletion: *Slow Docker command* is not considered
- **TPS is not deterministic**

New Scheduling Algorithm

Autonomous Container Scheduling

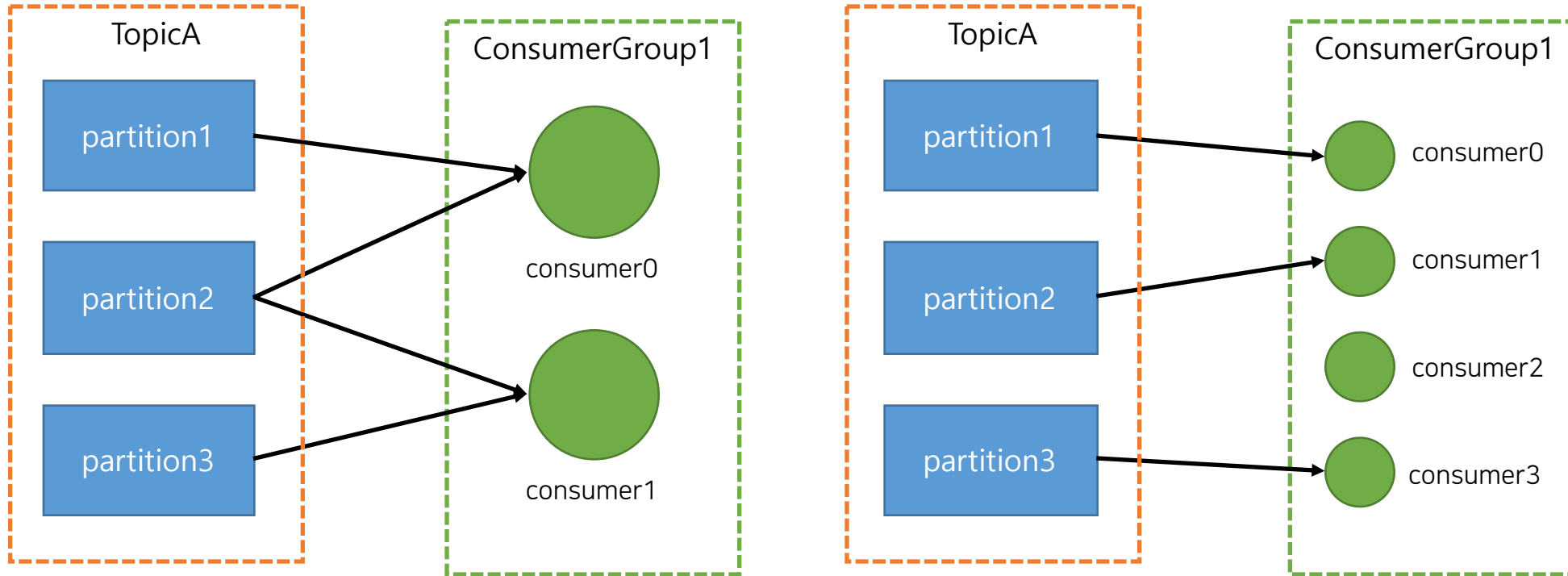
Kafka Partitions

- *Partition* is a unit of parallelism
 - Read/Write
 - Replication
- Kafka message has key and Kafka supports *pluggable partitioner*.
 - Message with same key can be sent to same partition



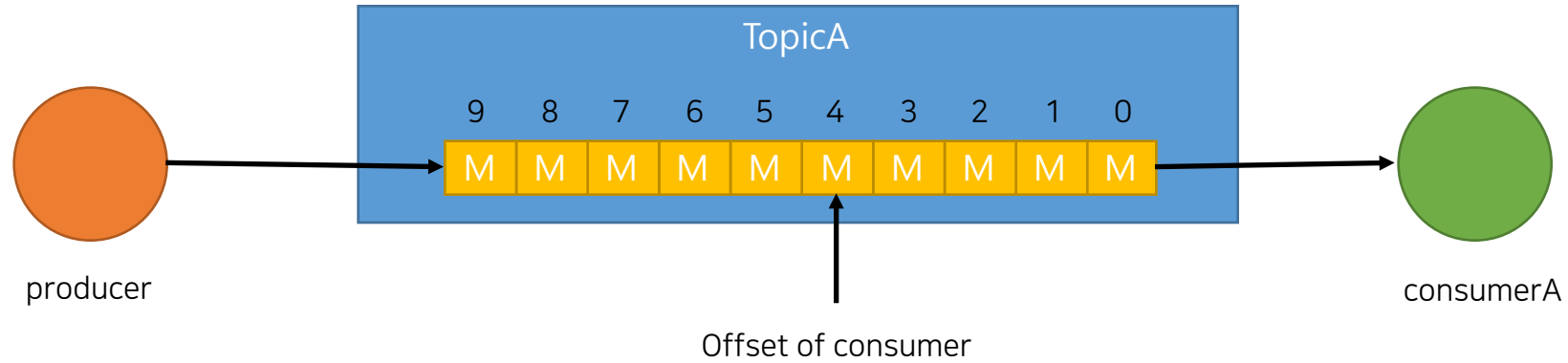
Kafka Consumer Group

- **ConsumerGroup** is a set of consumers which point to same topic
 - Each consumers can read message from all partitions
 - Based on the number of consumers, assigning plan is changed.
 - *Maximum parallelism is limited to the number of partitions*



Kafka Consumer Lag

- *ConsumerLag* means the number of not processed messages for the given consumer



ConsumerLag: 5

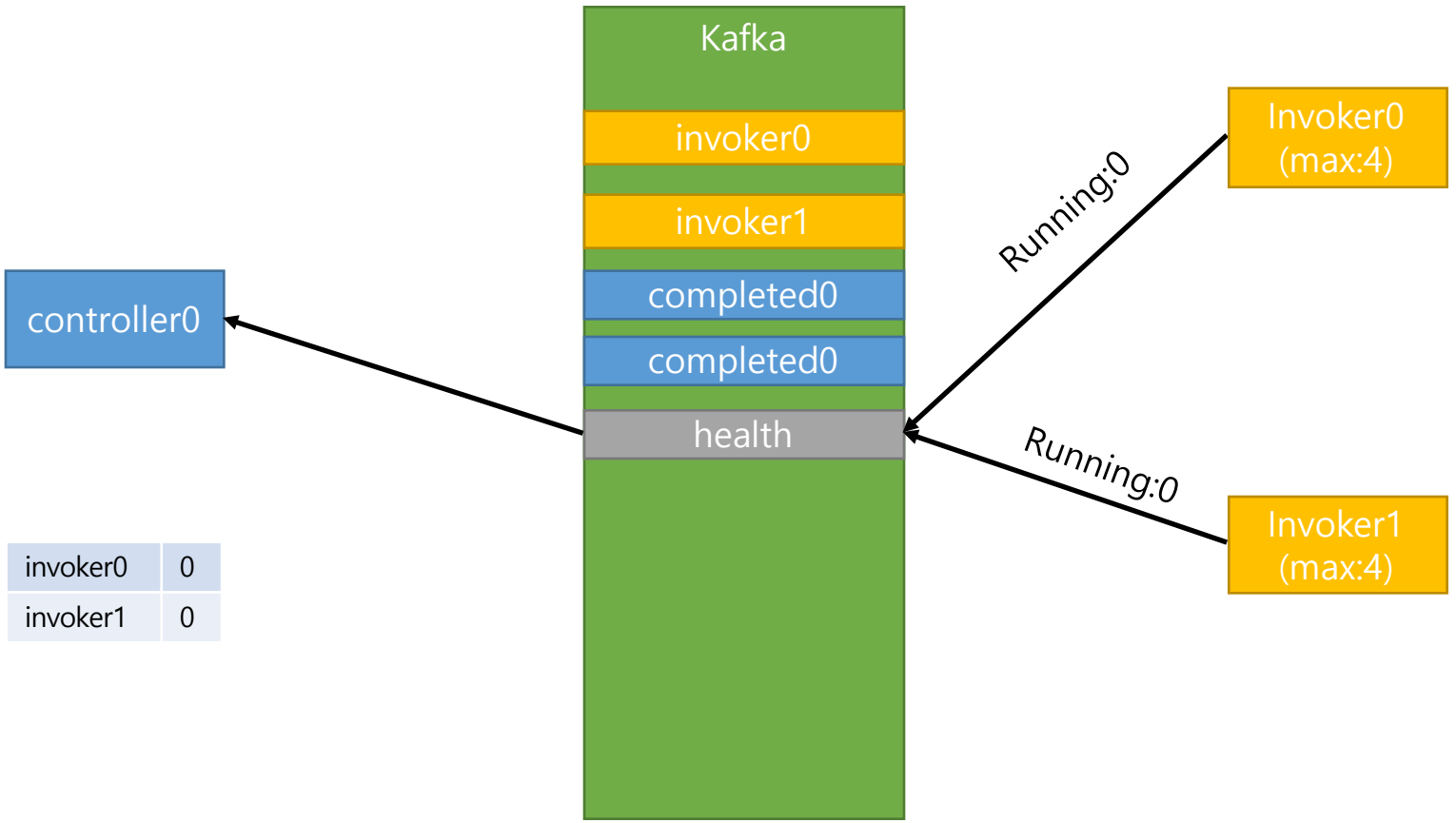
Autonomous Container Scheduling – Basics

1. Each actions has its own topic.
2. Each container proxies directly fetches requests from its own Kafka topic.
3. Invokers only manage container creation/deletion.
4. A created container keeps running for the given time(30s ~ 1min).
5. All containers with same topic form a same *ConsumerGroup*.
6. Limit works in per-action basis.
7. Default limit is 1, so only one container is created for an action at first time.
8. As limit is increased, *# of partition* for the given topic is also increased.
9. If more TPS is required, *invoker create more containers* for the topic.
10. If consumer lag is high, controller respond with *429 Too many request* .

Autonomous Container Scheduling – Whole Flow

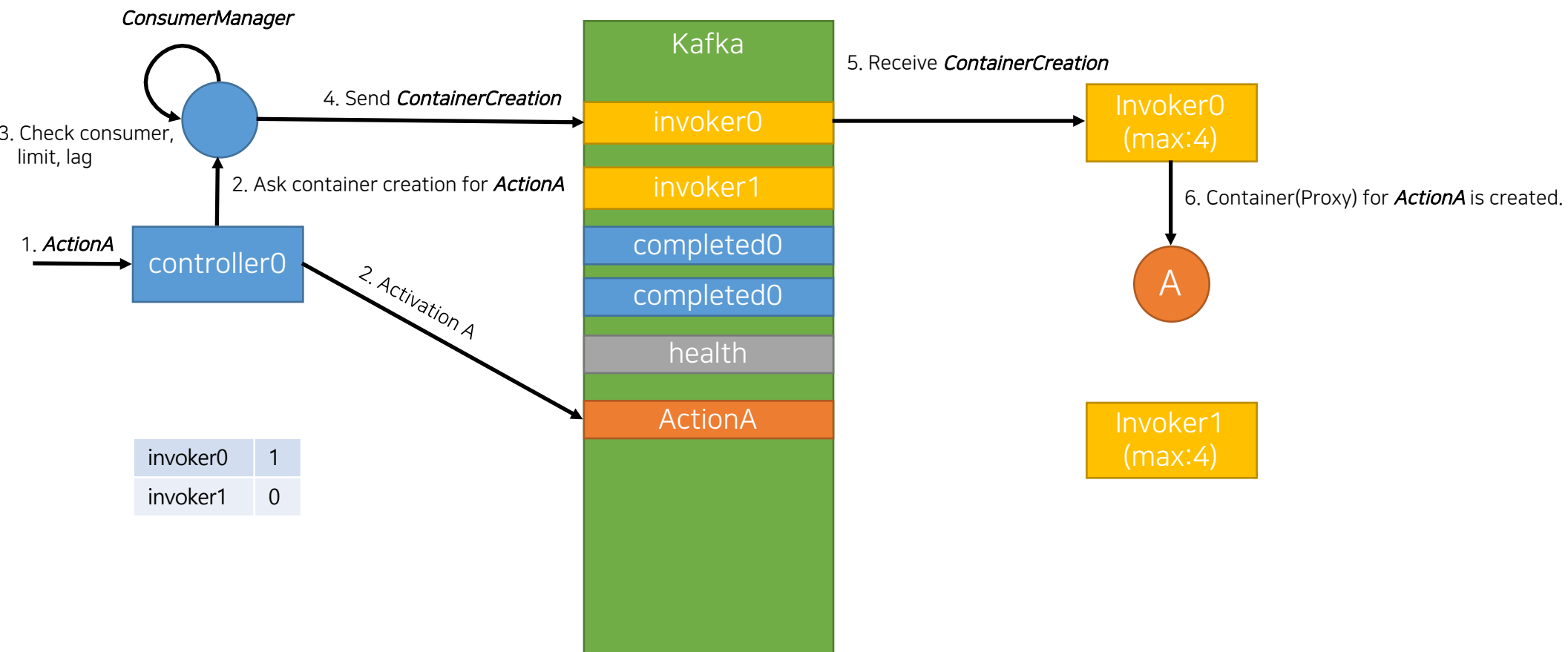
1. Each actions has its own Kafka topic(ex: *{namespace}-{name}*).
2. Parallelism of action is determined by *per-action limit* and *Kafka partitions*.
3. Each Invokers sends it's status via Health message (every 100~200ms).
4. All controllers keep slots for all invokers.
5. Once request comes, controller checks whether consumer exists for the topic(*{namespace}-{name}*).
 1. If exists, controller compares *# of consumers* , *action limit*, checks *consumer lag* and sends request directly to the *action topic*.
 2. If not exist or consumer lag is high, controller sends request directly to the *action topic*, at the same time, *sends container creation request* to invoker with **least loads**.
 3. If *# of consumers* = *action limit*, and *consumer lag* is too high, respond back with **429 Too many requests**
6. Once invoker receives container creation message, it creates a container for the given action and initialize it.
 1. If *busyPool* reaches *maxPoolSize*, it sends failed message back to the controller, then controller reschedules that message to other invokers.
 2. If already *# of consumers* = *action limit*, it sends *SkipMessage* back to the controller.
7. Once a container is running *ContainerProxy* directly reads message from Kafka and execute codes and send results to controller via Kafka.
8. Multiple containers for a specific actions belong to same *ConsumerGroup({namespace}-{name})*.
9. If limit is changed, *# of partitions* for the topic should be changed as well. (*# of partitions* = *# of concurrent containers*)
10. Optimization: If consumer lag is small enough, new container is not created, though *# of consumers* < *# of partitions*
11. A container does *not pause or removed* though execution is finished for 30s ~ 1min.
12. If no request comes for 30s ~ 1min, invoker terminates/pauses that container.

Autonomous Container Scheduling: Invoker status via health message

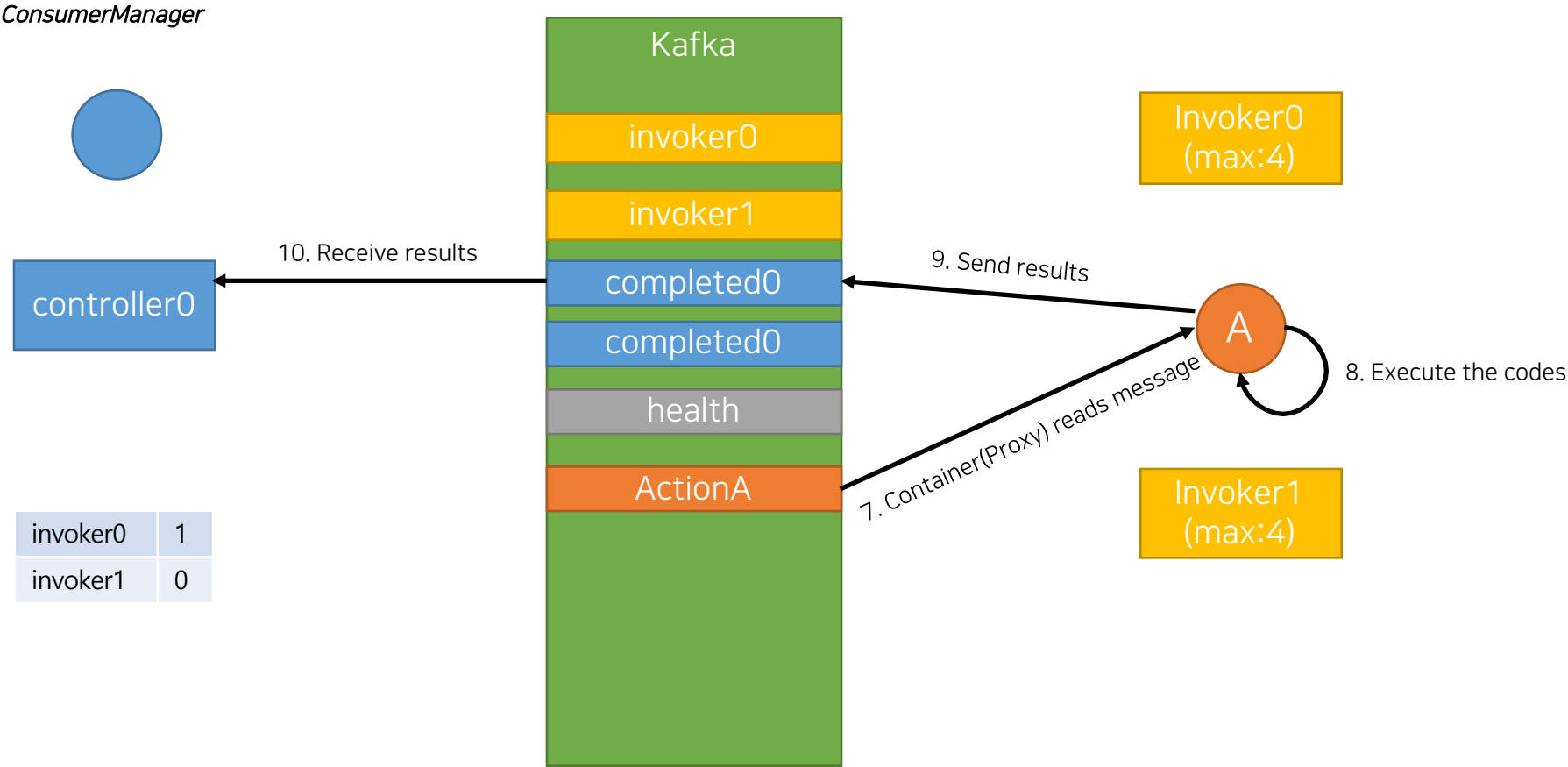


Each invokers send its pool status via health messages

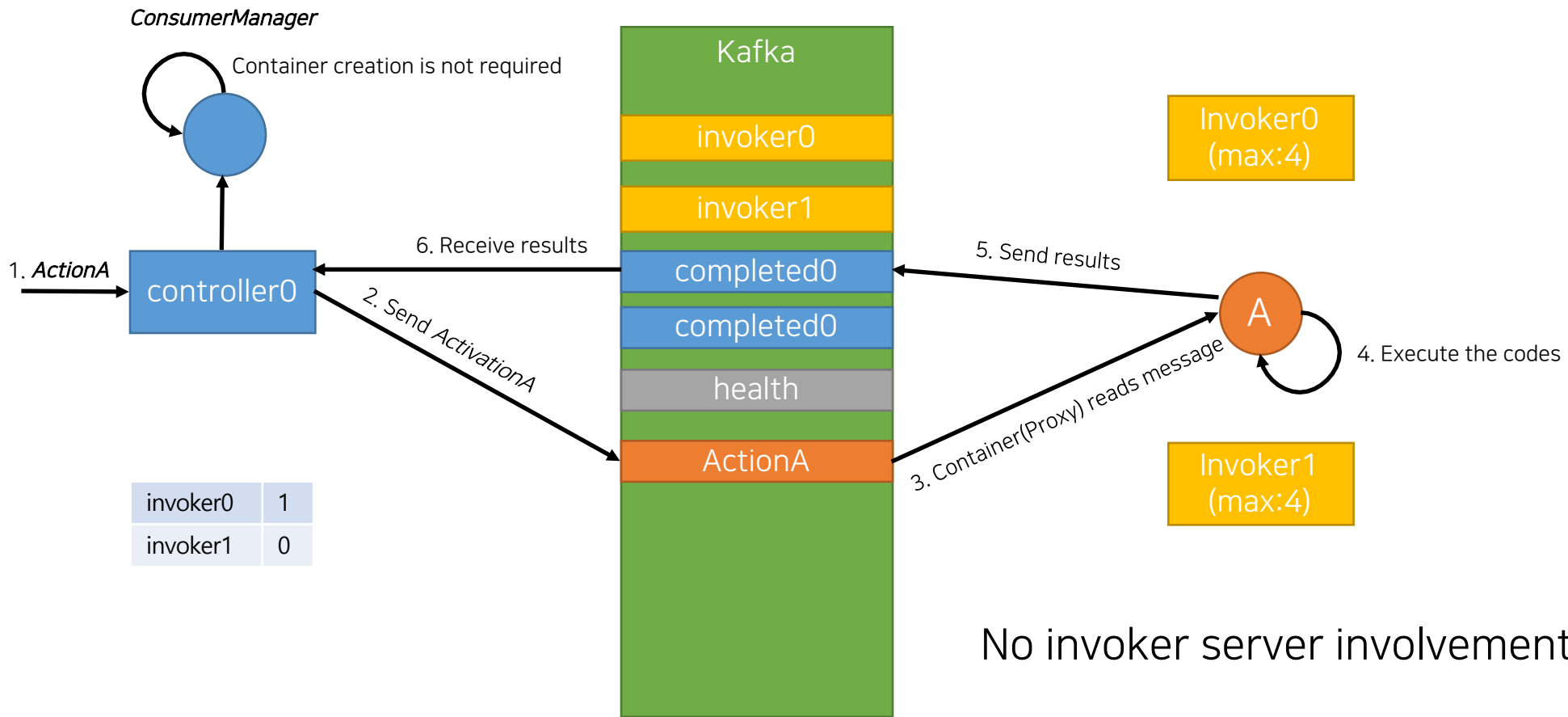
Autonomous Container Scheduling: Basic Flow (1/3)



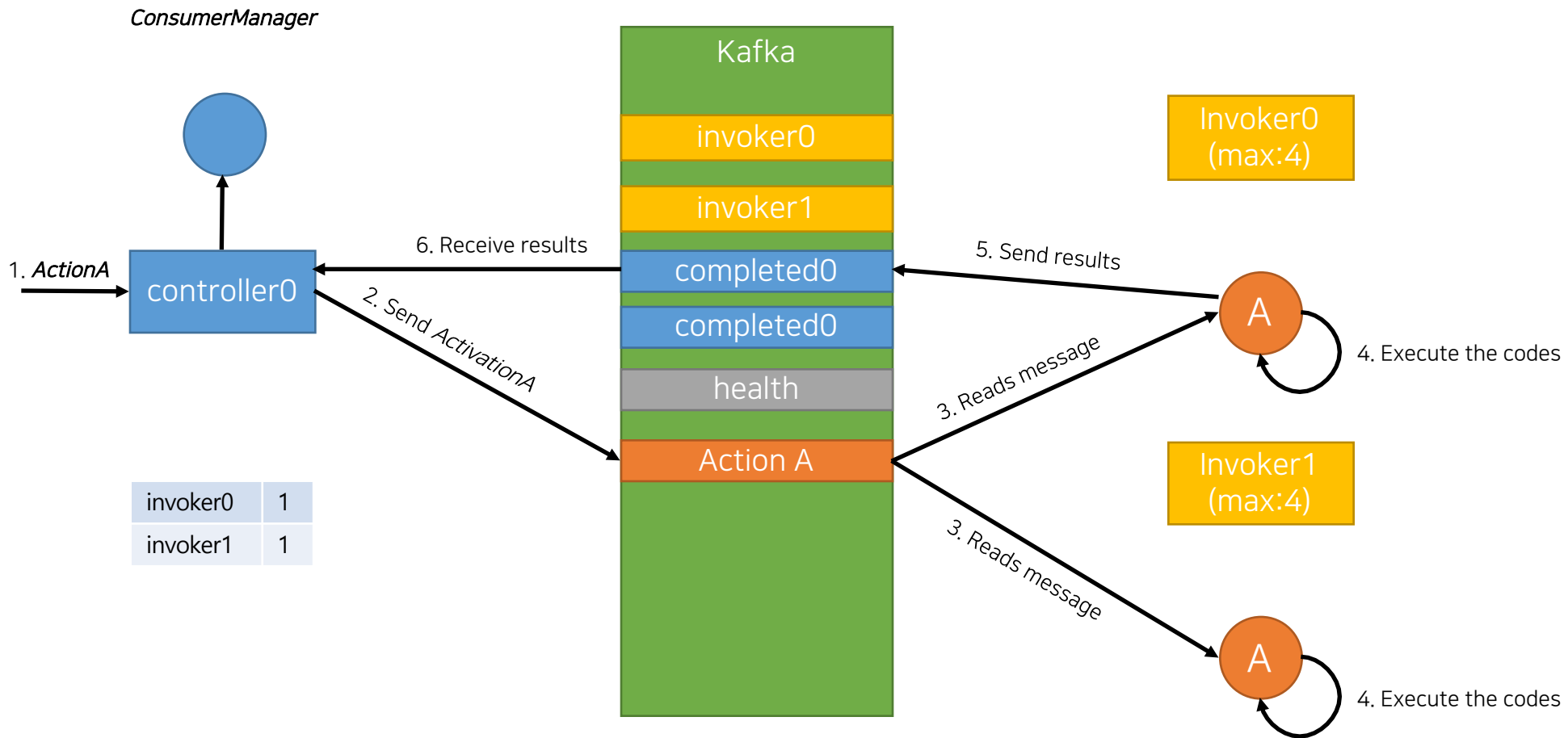
Autonomous Container Scheduling: Basic Flow (2/3)



Autonomous Container Scheduling: Basic Flow (3/3)

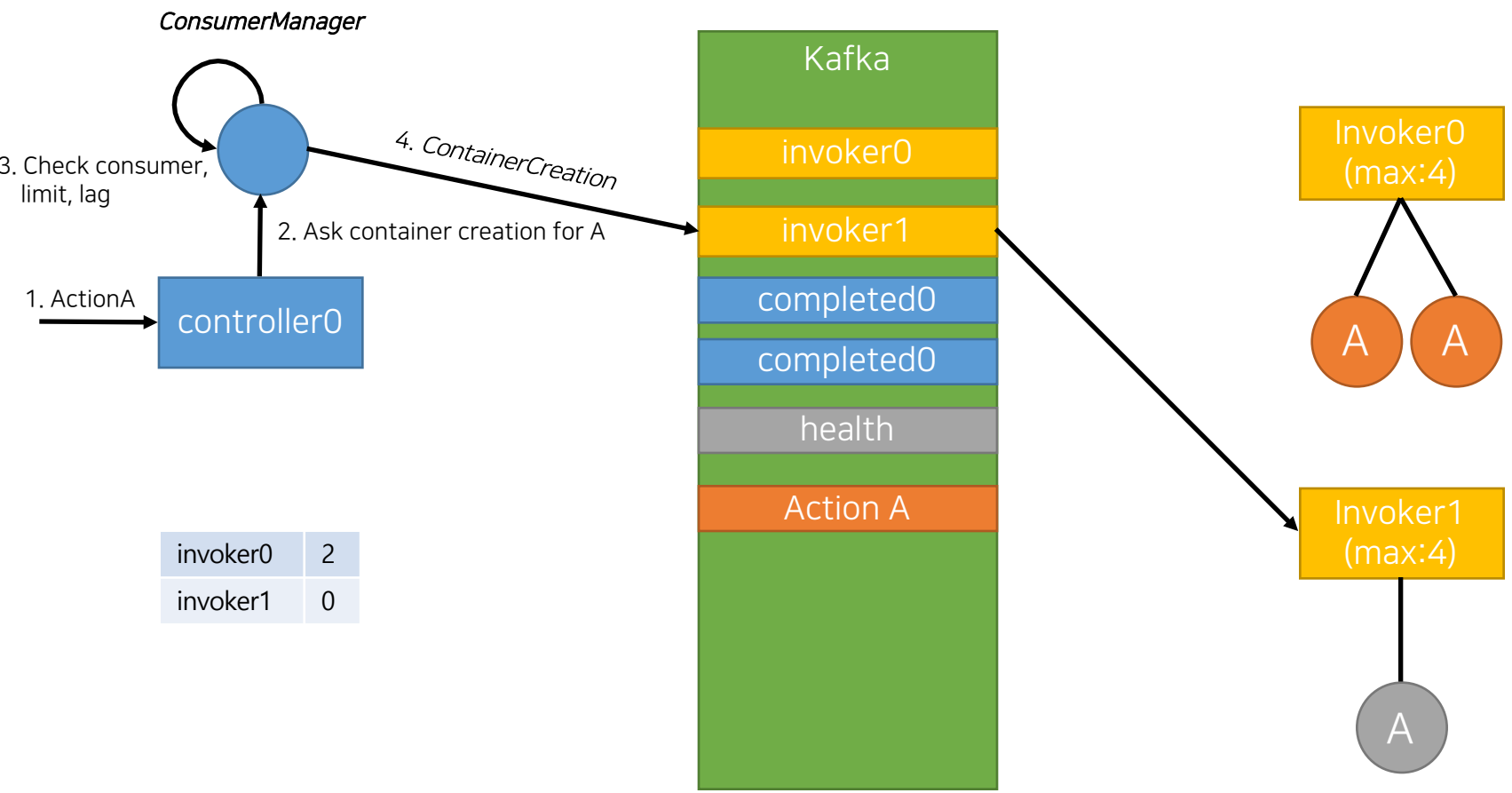


Autonomous Container Scheduling: Container Location Free Scheduling



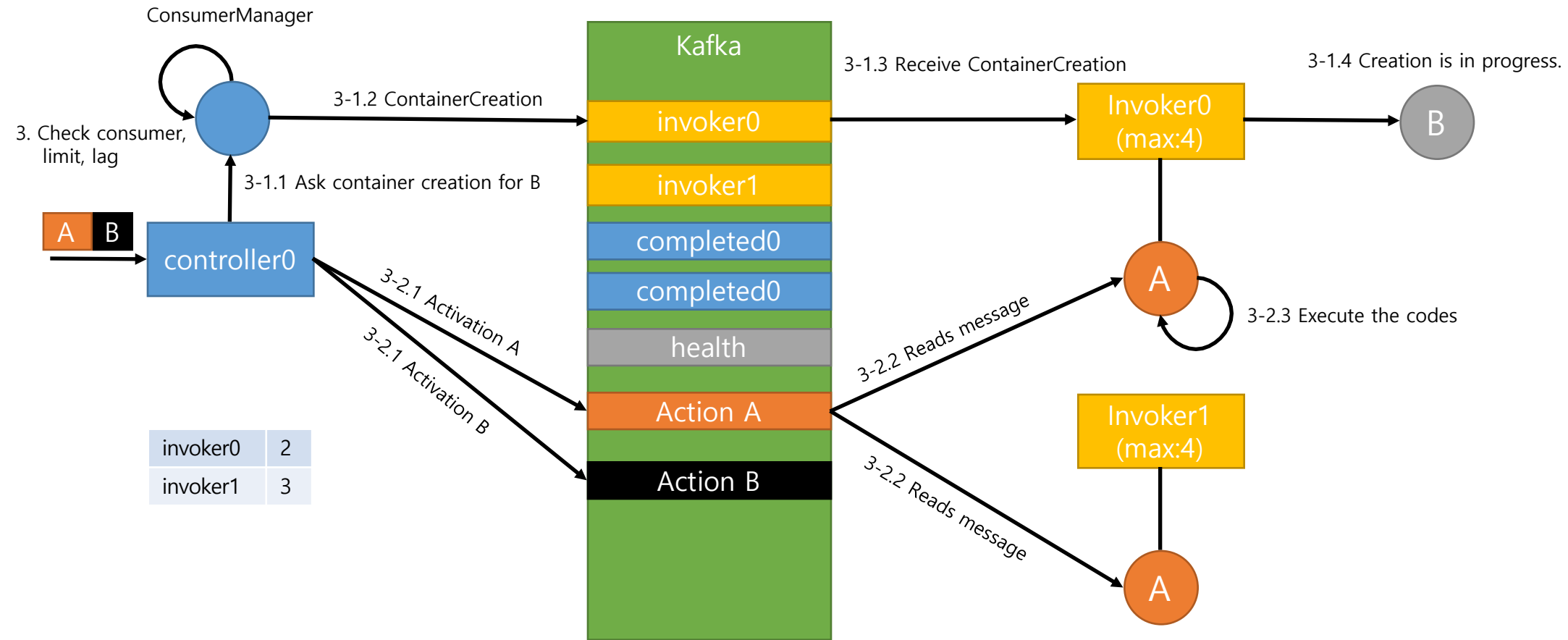
No need to send requests to the same invoker.
Controllers can just send requests to action topics.

Autonomous Container Scheduling: Even load distribution



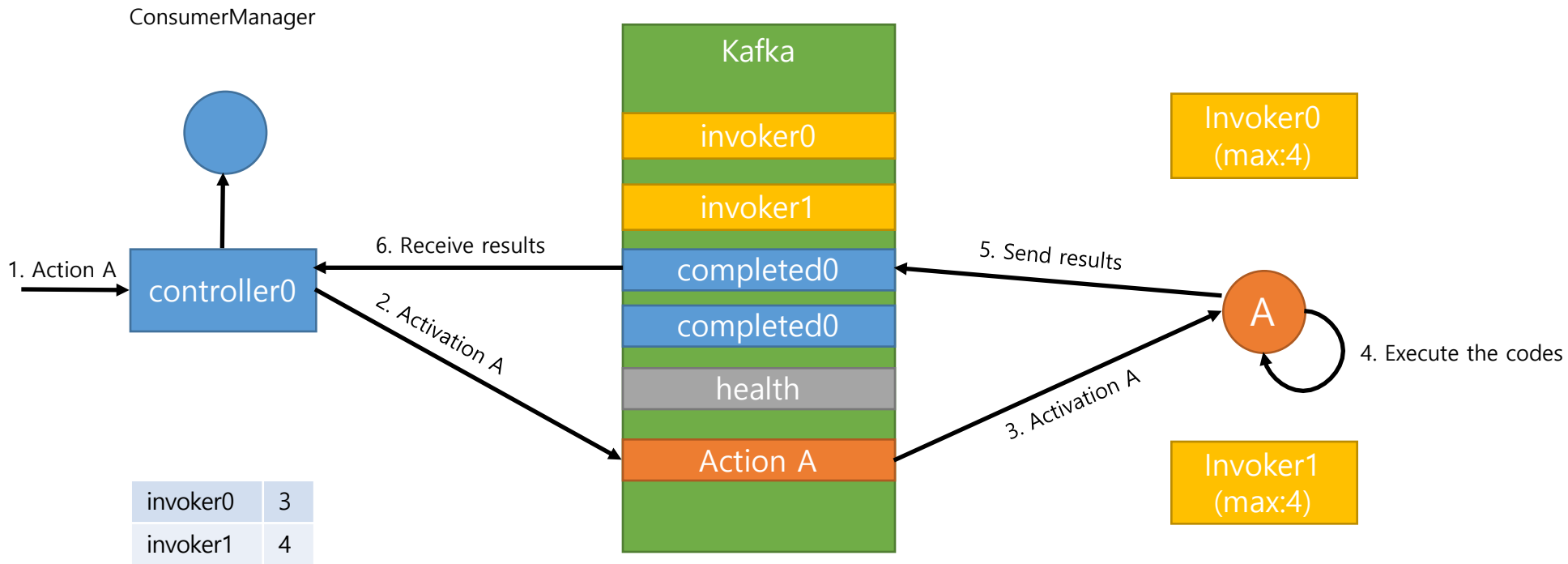
No need to consider the location of existing containers.
It can just send *ContainerCreation* to an invoker *with the least loads*.

Autonomous Container Scheduling: Performance Isolation



Container creation does not affect the existing container performance

Autonomous Container Scheduling: Improved Base TPS

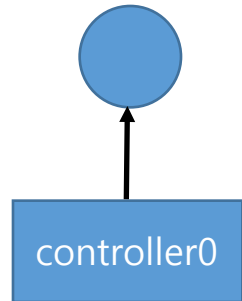


No other logics involved.
Just reads the message and execute it.
Base TPS of one container increases.

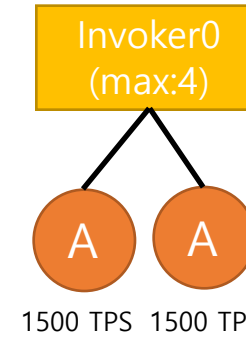
I got about 1,500TPS with only 1 container

Autonomous Container Scheduling: Deterministic TPS

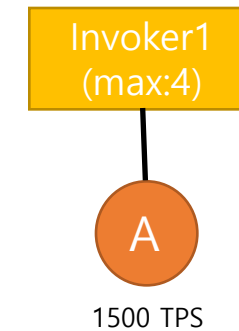
ConsumerManager



invoker0	2
invoker1	3



Total TPS for Action A: 4,500
For 6,000 TPS -> Add one more container.

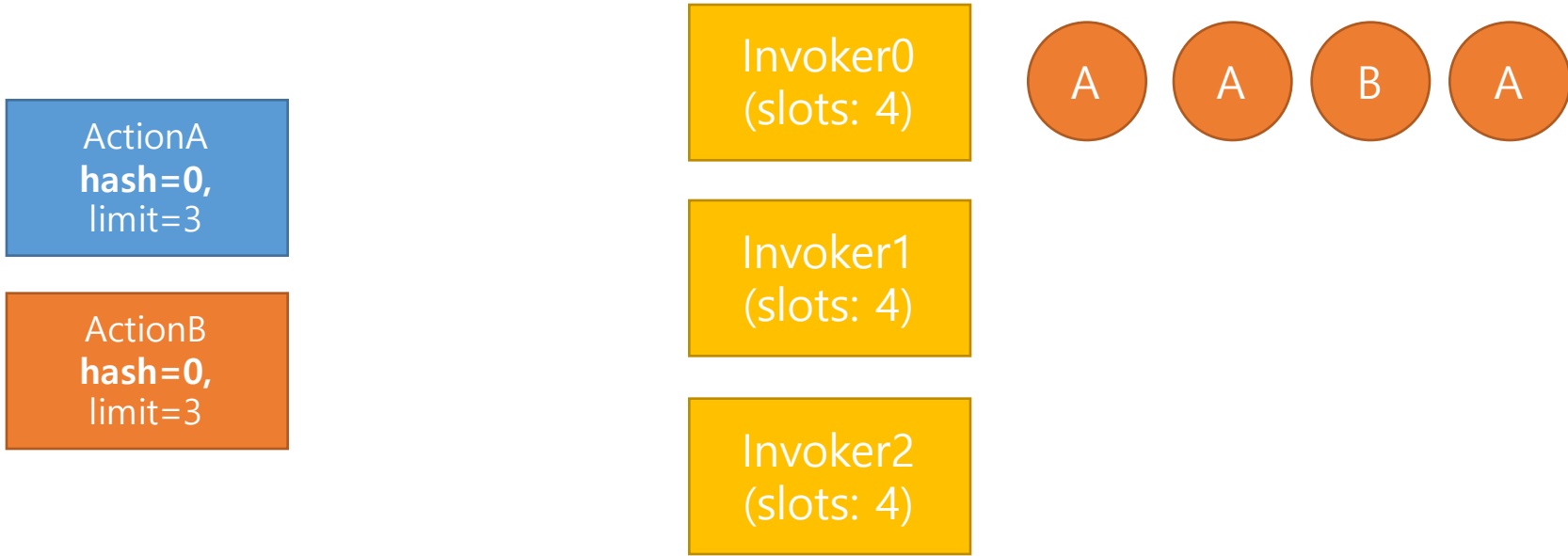


Based on performance isolation, TPS becomes deterministic.
Target TPS can be achieved deterministically

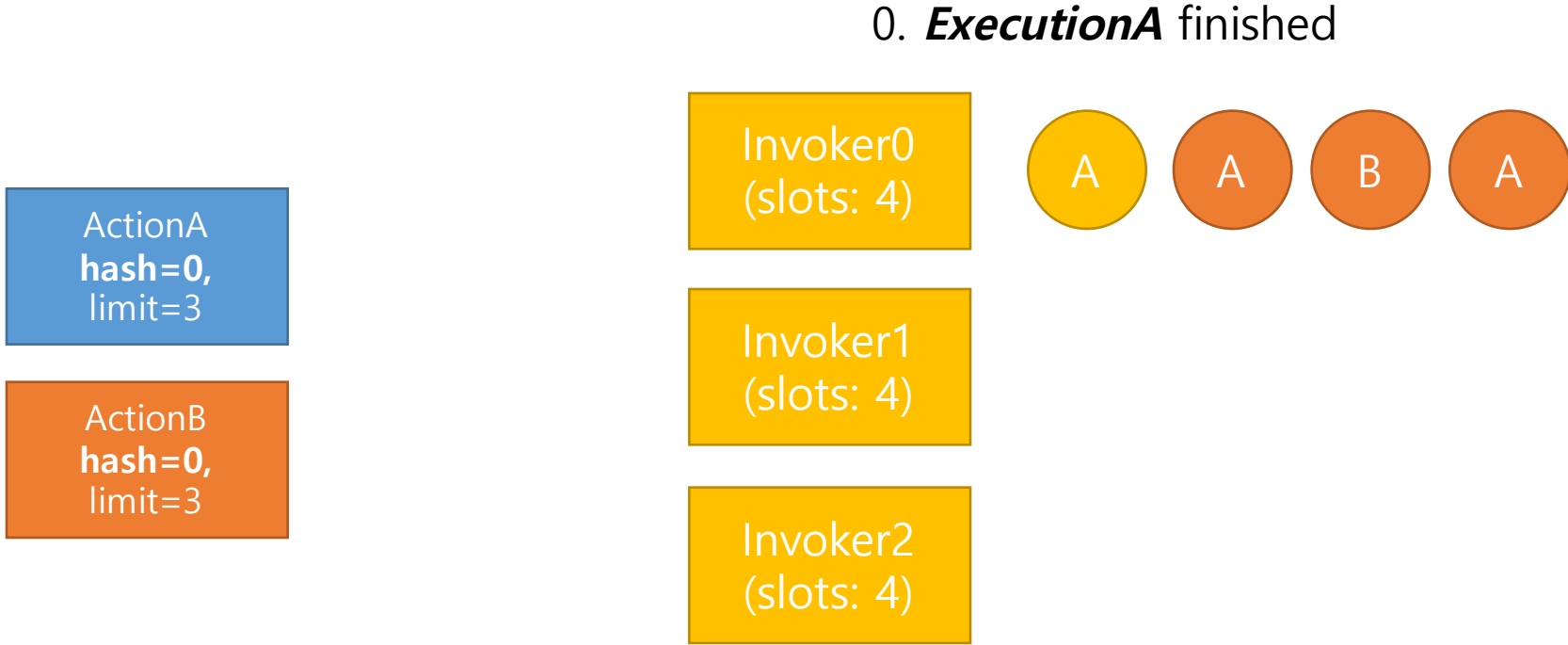
(Note: In real world, containers share host resources, TPS may not be linearly increased. But anyway, TPS would be much more predictable)

Review previous Issues
with new Scheduling Proposal

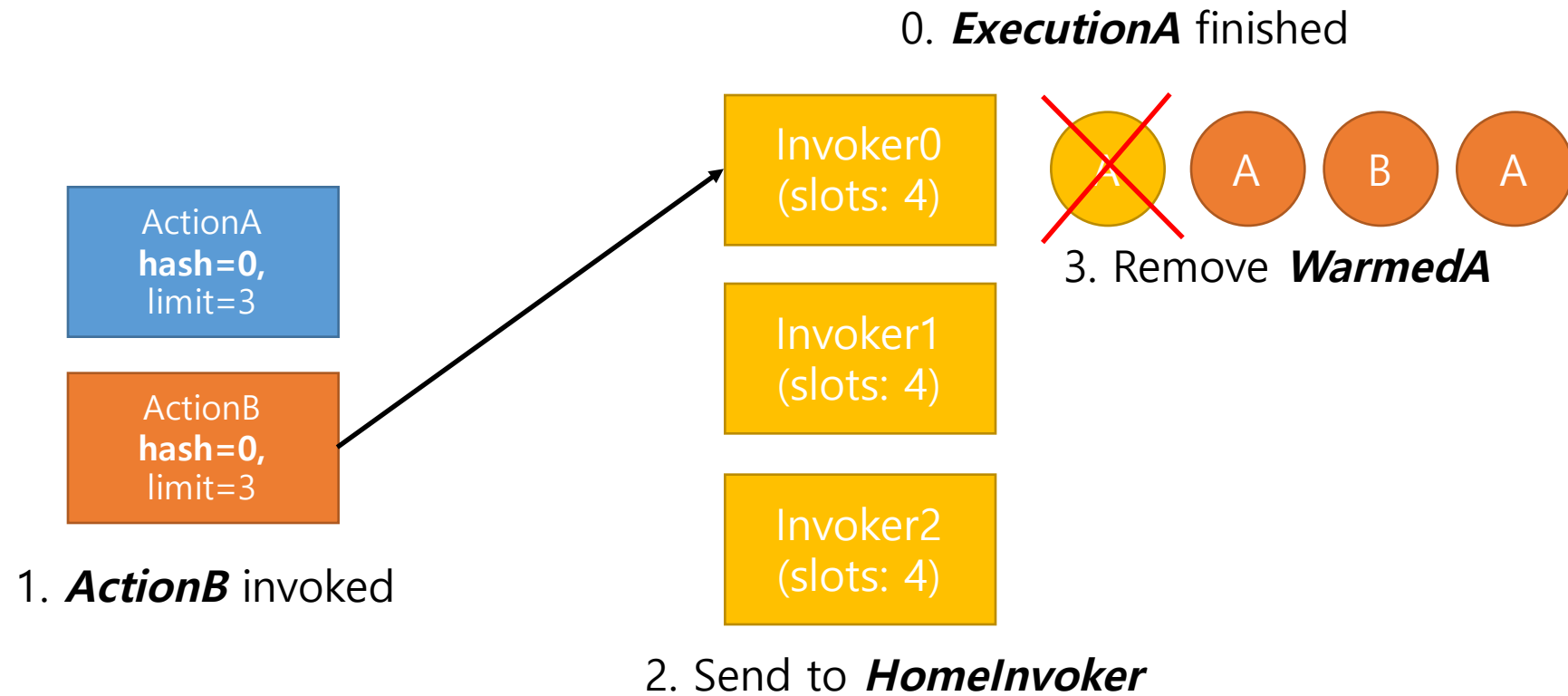
Problems in real scene – worst case: intervention of actions



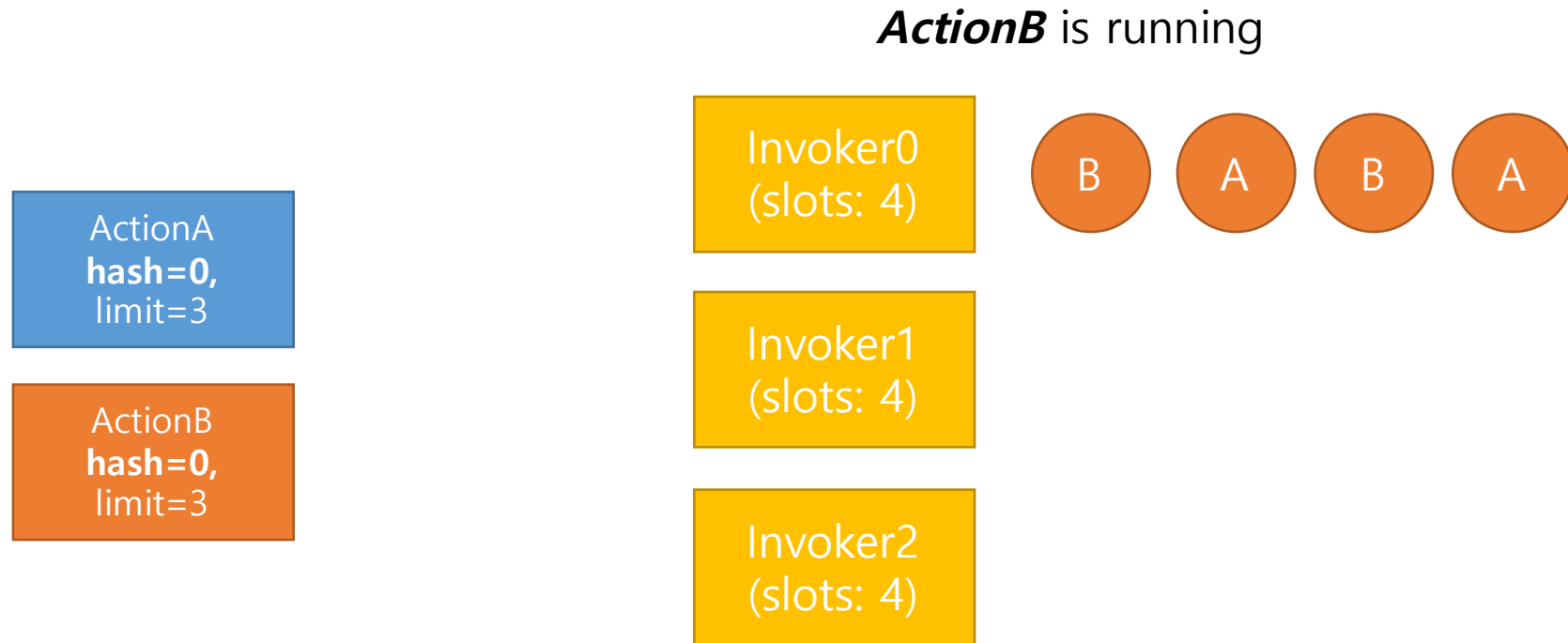
Problems in real scene – worst case: intervention of actions



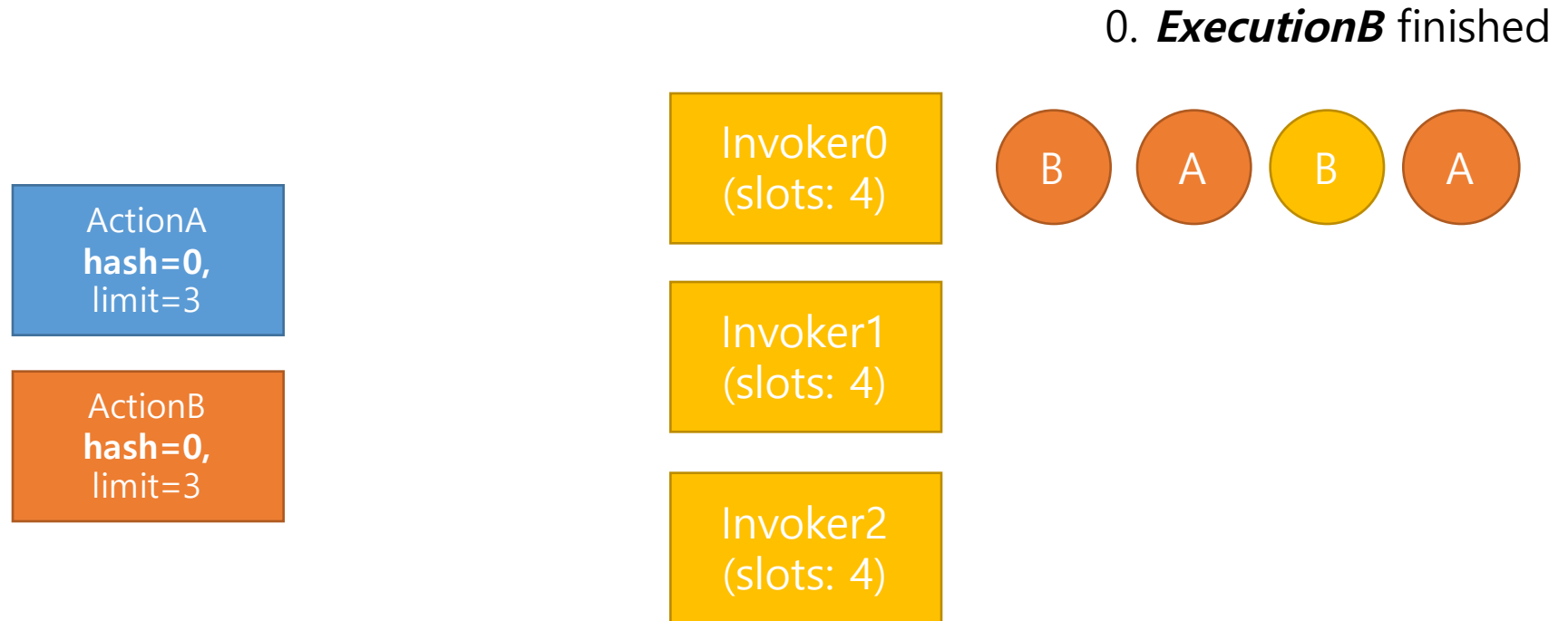
Problems in real scene – worst case: intervention of actions



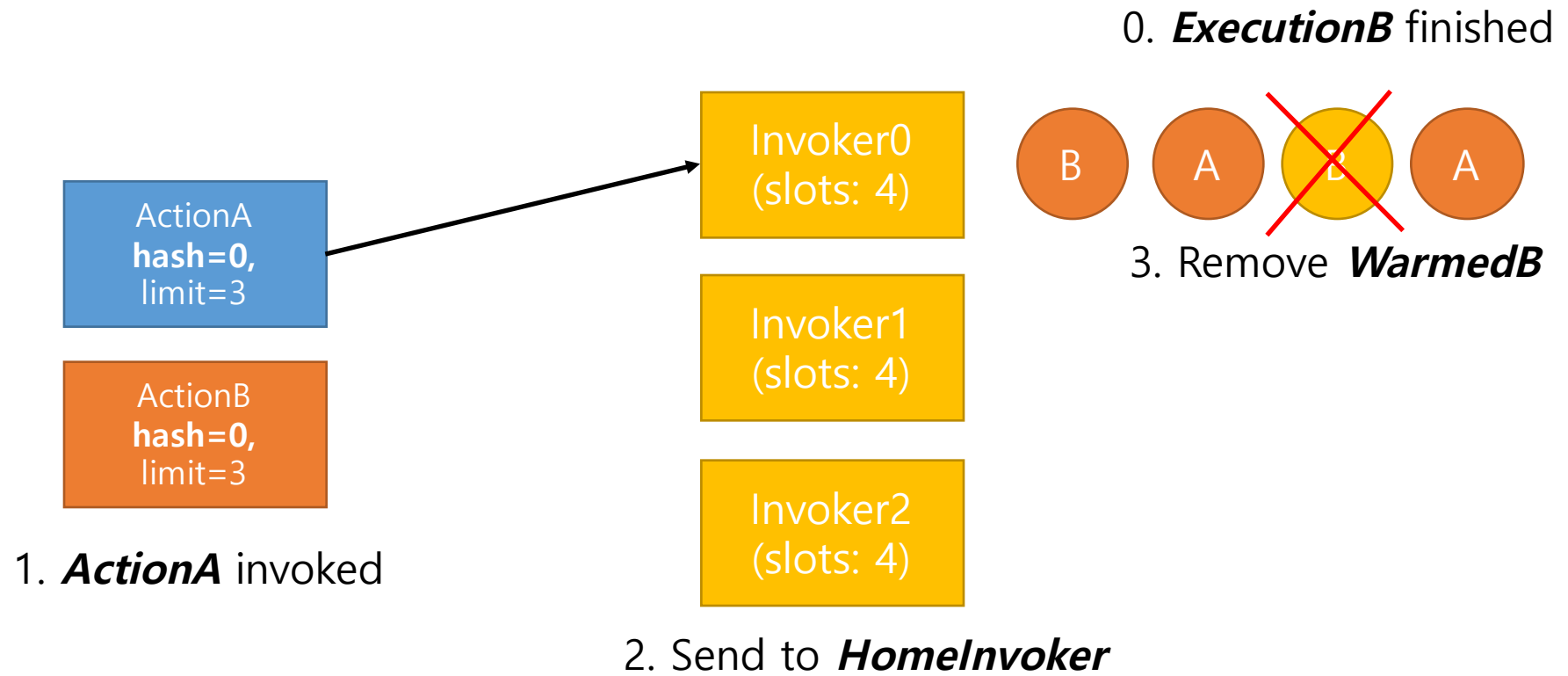
Problems in real scene – worst case: intervention of actions



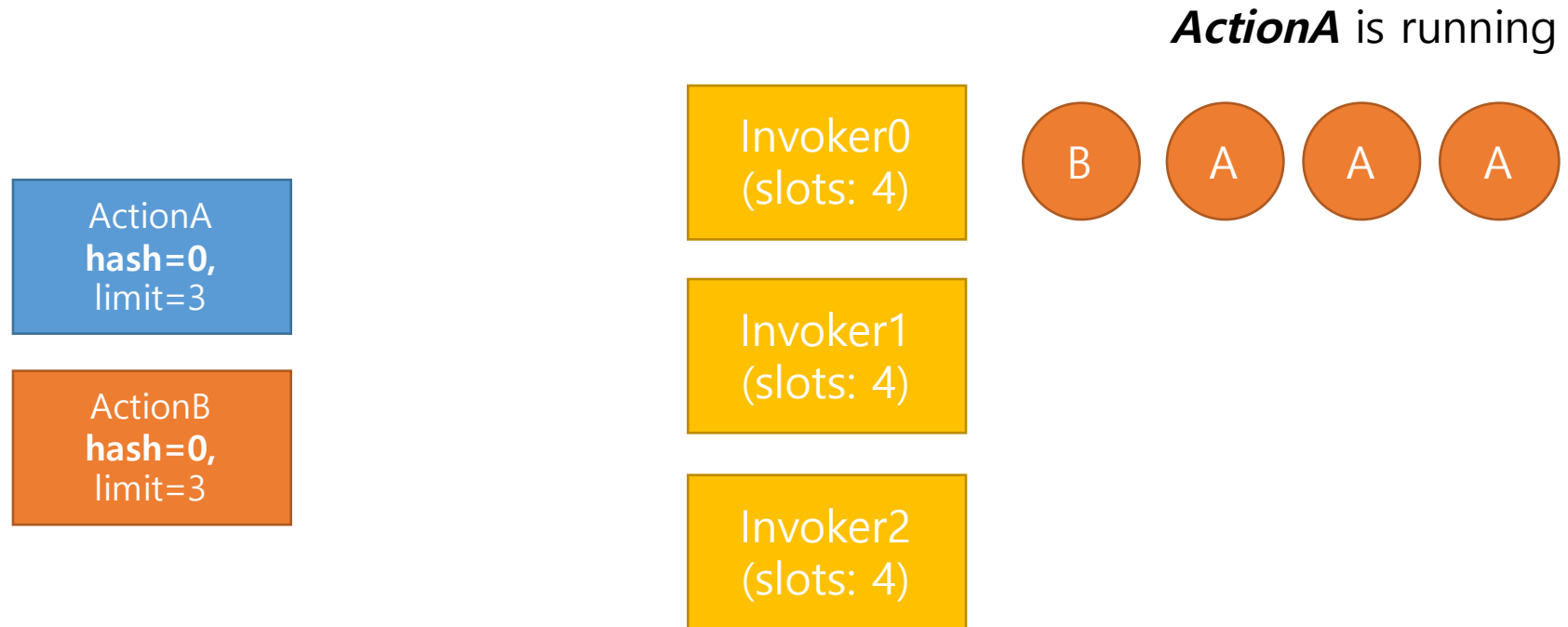
Problems in real scene – worst case: intervention of actions



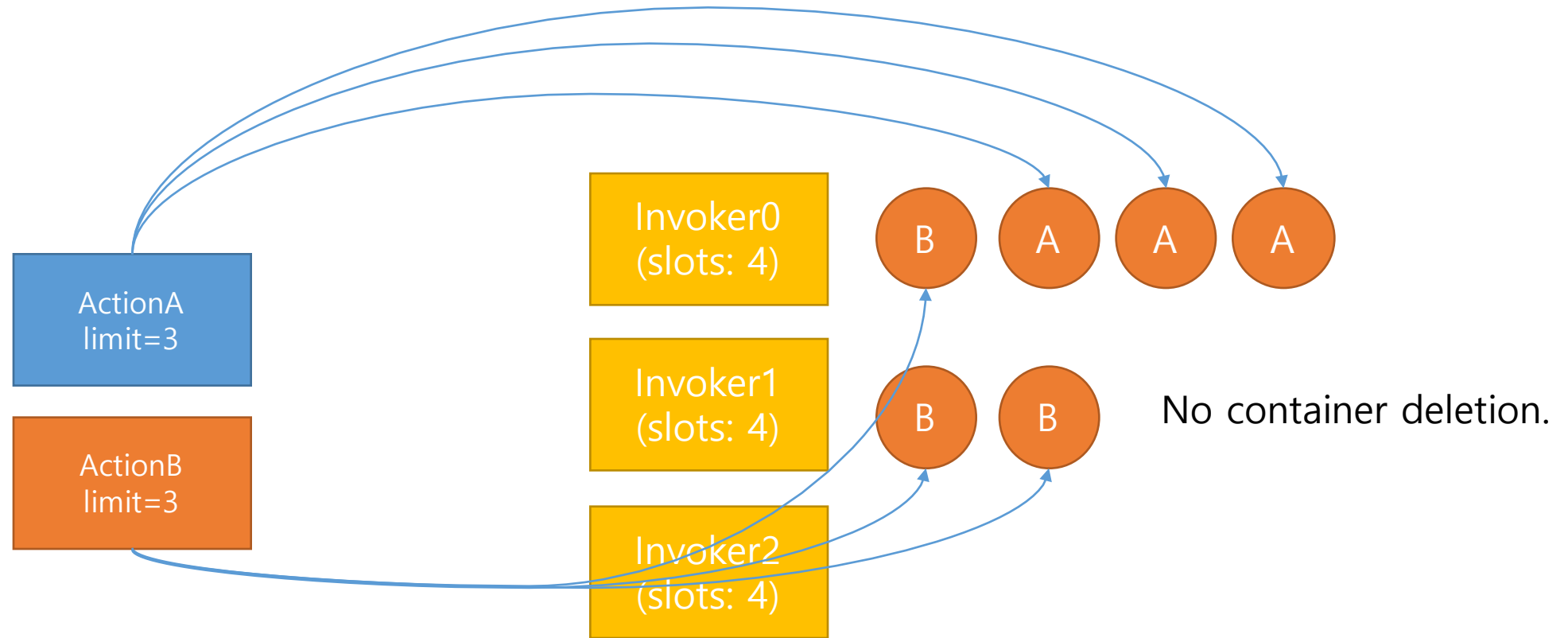
Problems in real scene – worst case: intervention of actions



Problems in real scene – worst case: intervention of actions

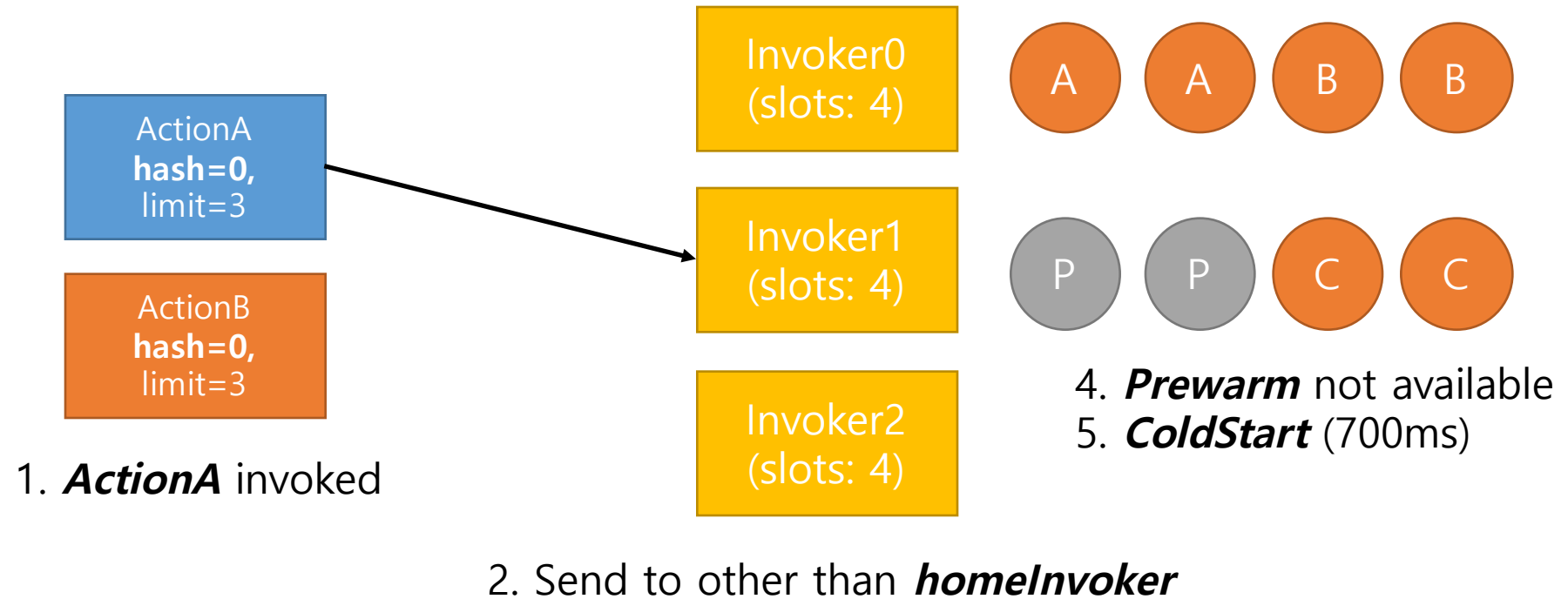


With new Scheduling Algorithm: Intervention of actions



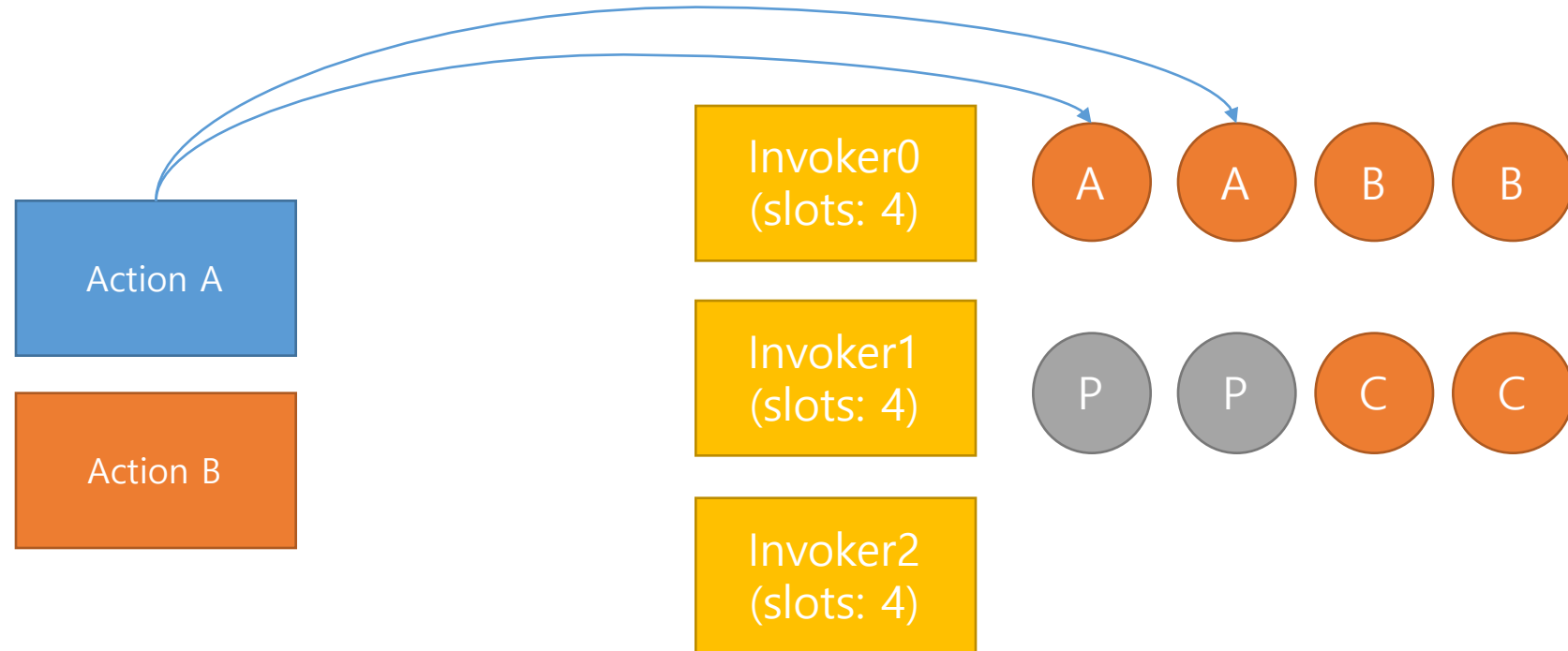
Containers can run on any invokers and directly read messages from Kafka.

Problems in real scene – worst case2: does not wait for completion



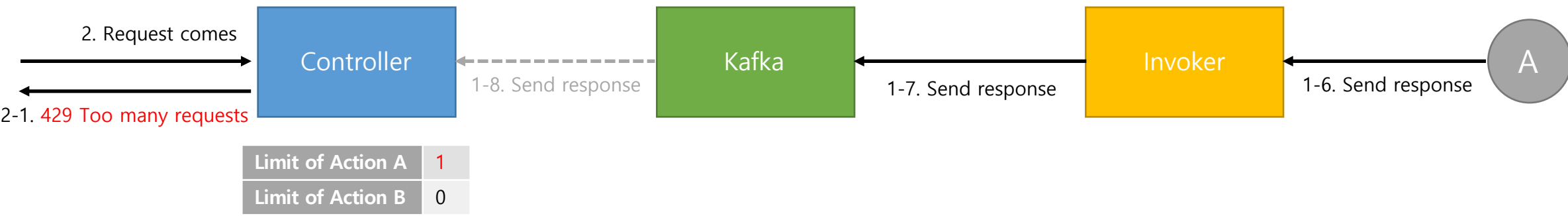
If execution time of **ActionA** is **20ms**, it takes **720ms** to run the code

With new Scheduling Algorithm: does not wait for completion

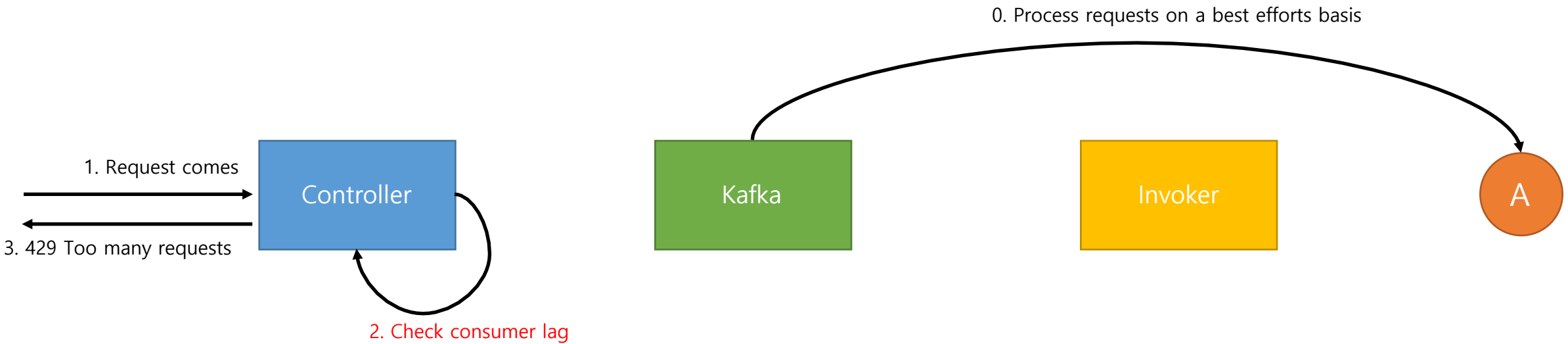


Inherently wait for previous execution.

Problems in real scene - action concurrent execution limit



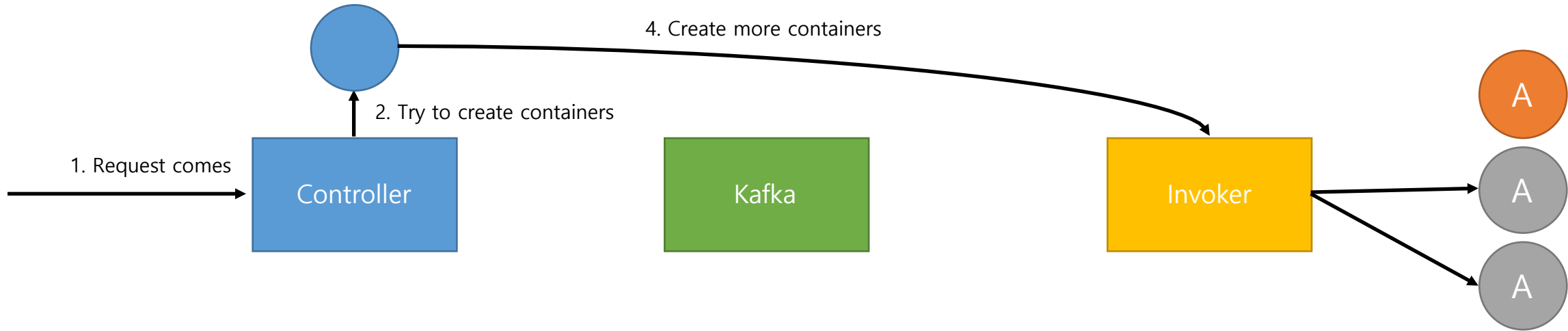
With new Scheduling Algorithm: action concurrent execution limit



Throttling is performed based on real processing speed

Problems in real scene - action concurrent execution limit

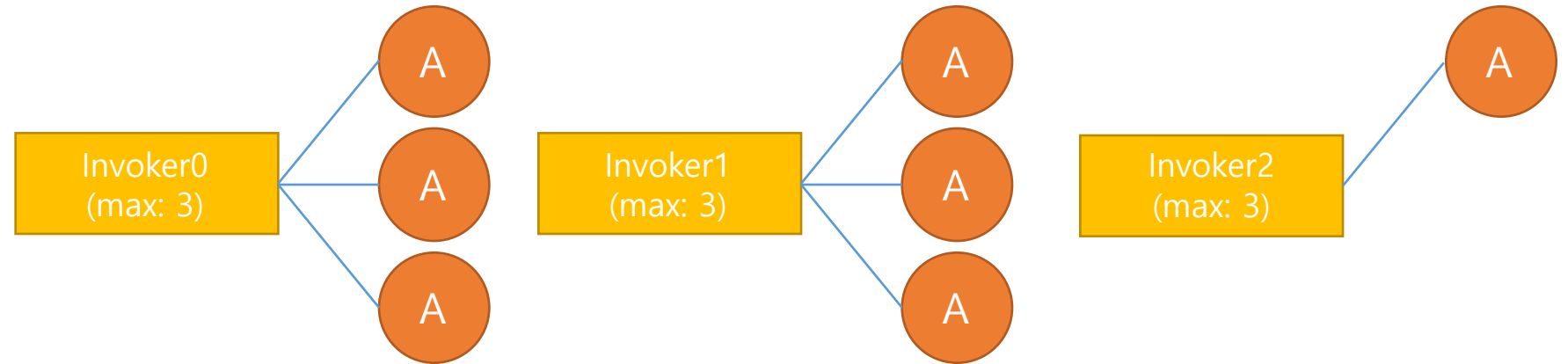
3. Compare # of consumers and and limit



Exactly same number of containers with limit is guaranteed under heavy loads

More fine-grained control over resources

Problems in real scene - action concurrent execution limit



MAX: 9
Current: 7



Need to add more invokers

Easy to figure out current resource status and decide to add more servers

Easy to resource planning

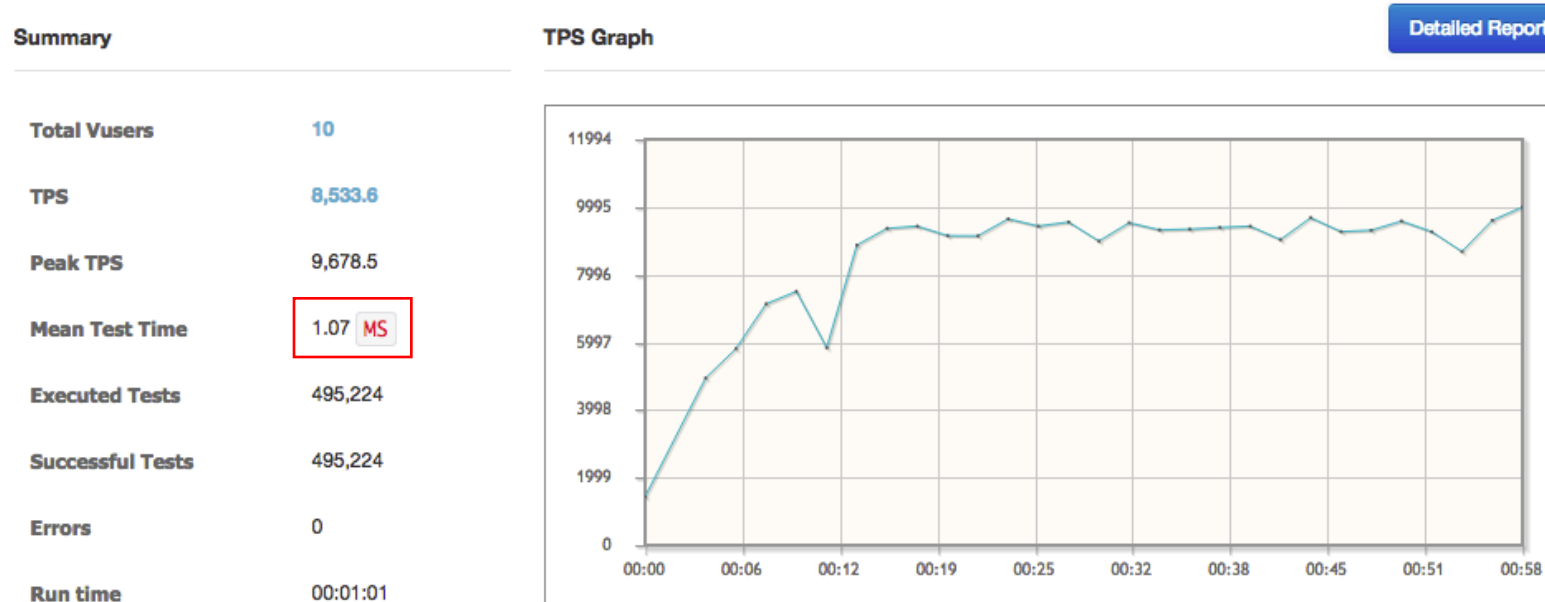
Pros and Cons of new Scheduling Proposal

- Pros
 - No need to send request to same invoker
 - No need to coordinate requests: *container itself read messages* from Kafka.
 - Can send less requests to the invoker.
 - Container *creation/deletion/pausing/resuming* is minimized.
 - *Request processing* is not affected by container *creation/deletion*.
 - Under max loads, exactly *same number of containers with limit* is guaranteed.
 - More fine-grained controller for *target TPS* (*Base TPS* × *# of containers*)
 - At some point, it's easy to figure out node addition is required or not. (*Easy resource planning*)

Pros and Cons of new Scheduling Proposal

- Cons
 - Controller should check *consumer lag* for every requests. -> increase execution time.
 - *Same number of topic* with the *number of actions* are required.
 - Action container can be *reserved for 30s ~ 1min*.
 - All runtimes should include *Kafka client*.
 - If limit changed, *# of partitions* should be also changed.

Cons - Controller should check consumer lag for every requests.



Took about **1ms** to check consumer lag

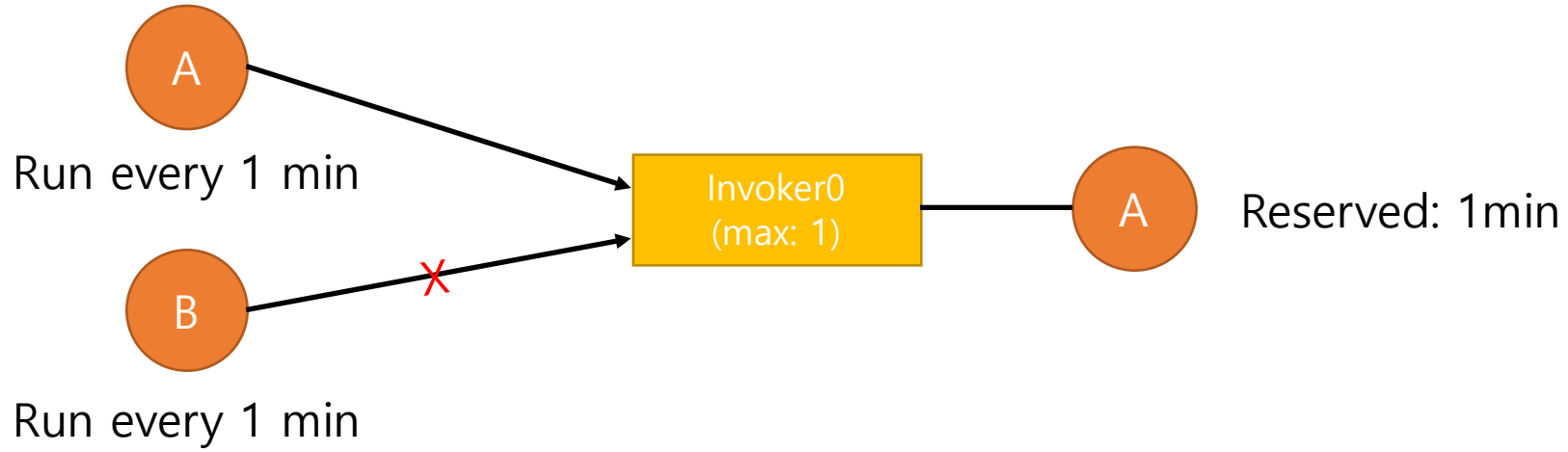
Cons - Same number of topic with the number of actions are required.

- The number of *active topics* will not be huge.
 - The number of active topics is limited to the maximum number of concurrent containers.
 - Benchmark results with 3 Kafka nodes:

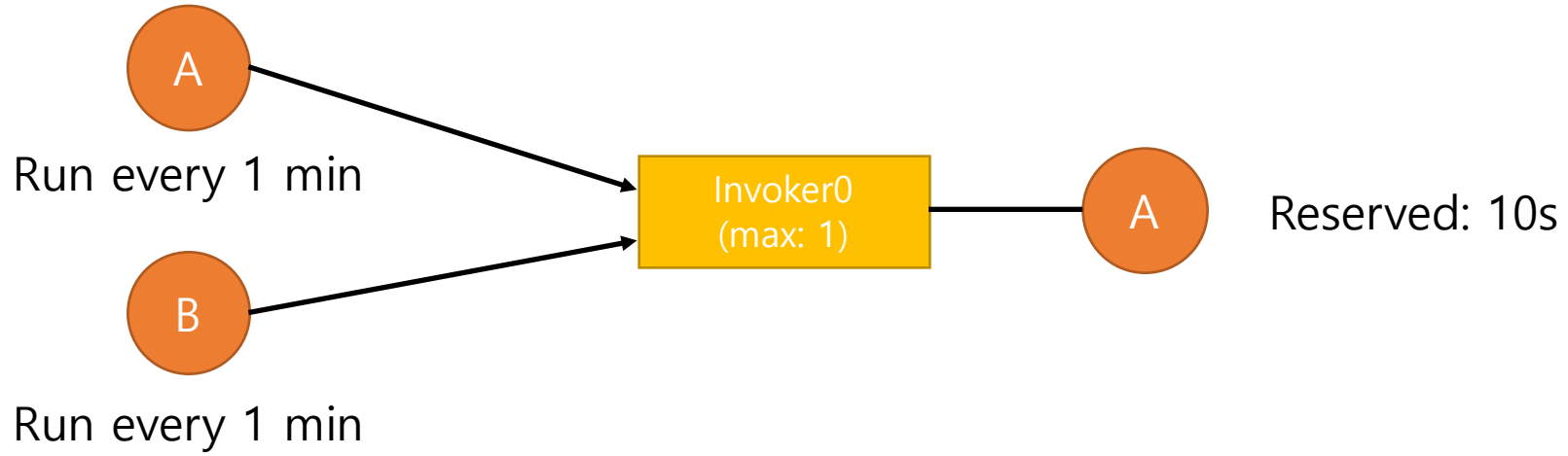
# of topics	Kafka TPS
50	34,488
100	34,502
200	31,781
500	30,324
1,000	30,855

- 1,000 active topics = 1,000 concurrent containers = **62.5 invokers**(8 cores, 10GB memory, MaxPoolSize=16)
 - If we need to have 62.5 invokers, surely we will have more number of Kafka nodes than 3 nodes.
 - Kafka nodes can be horizontally scaled-out.

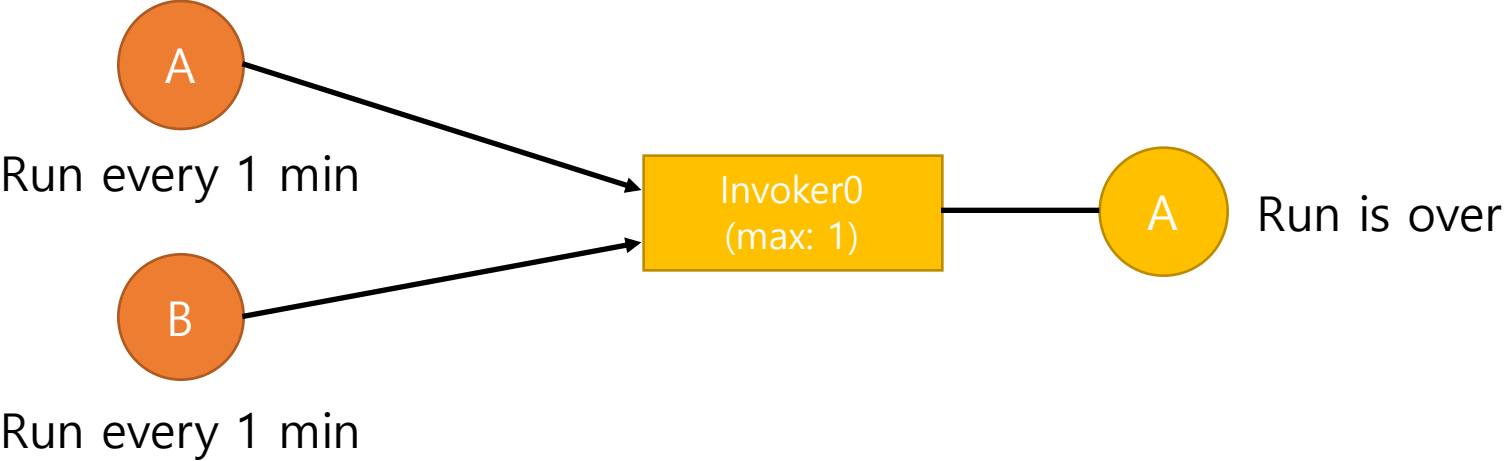
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



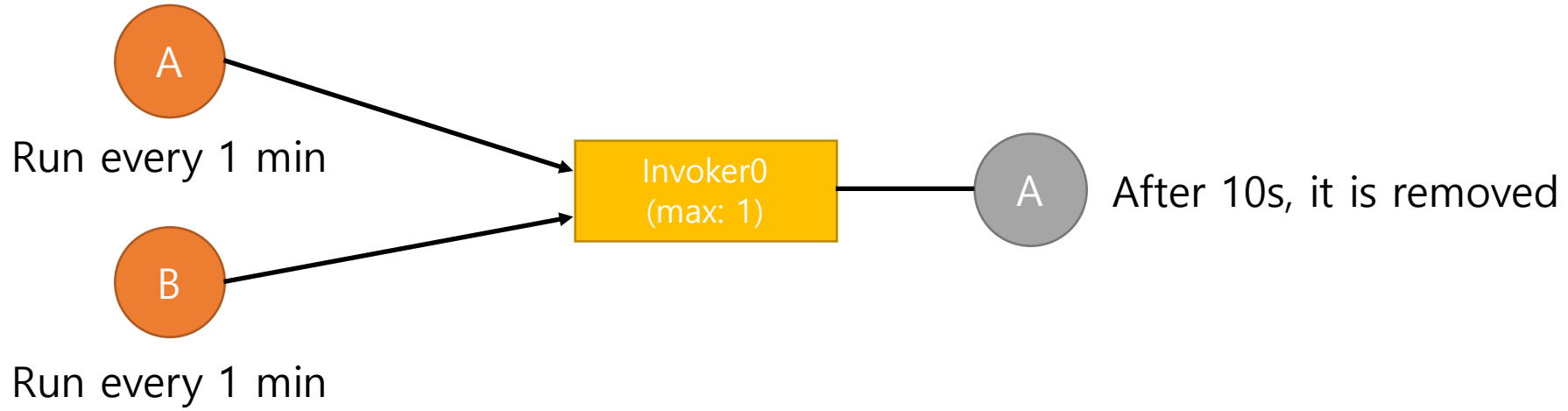
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



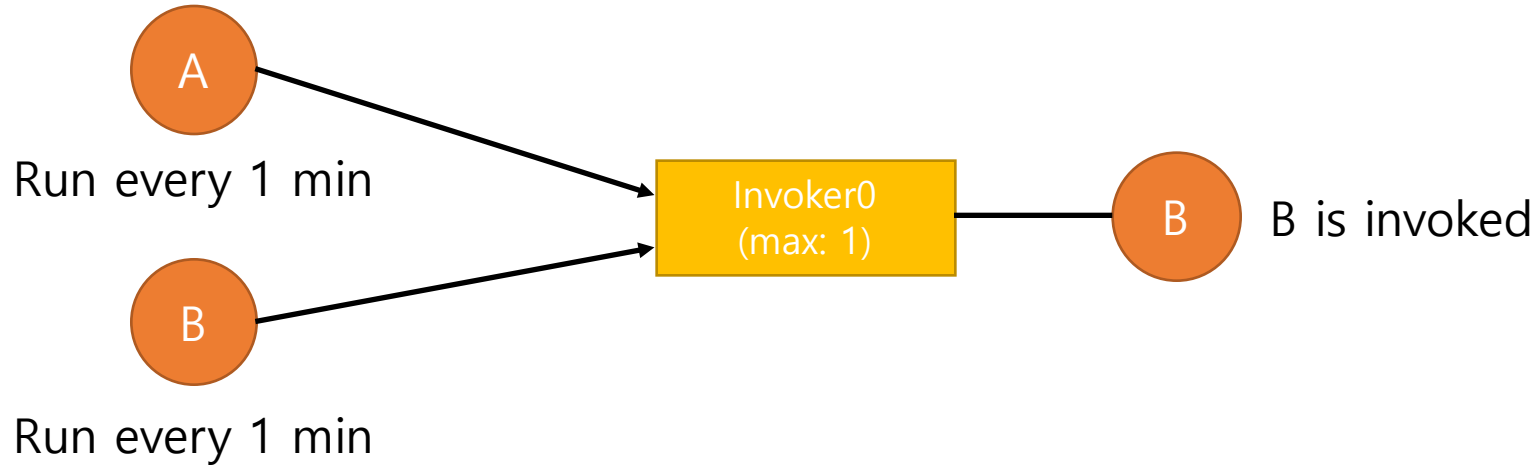
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



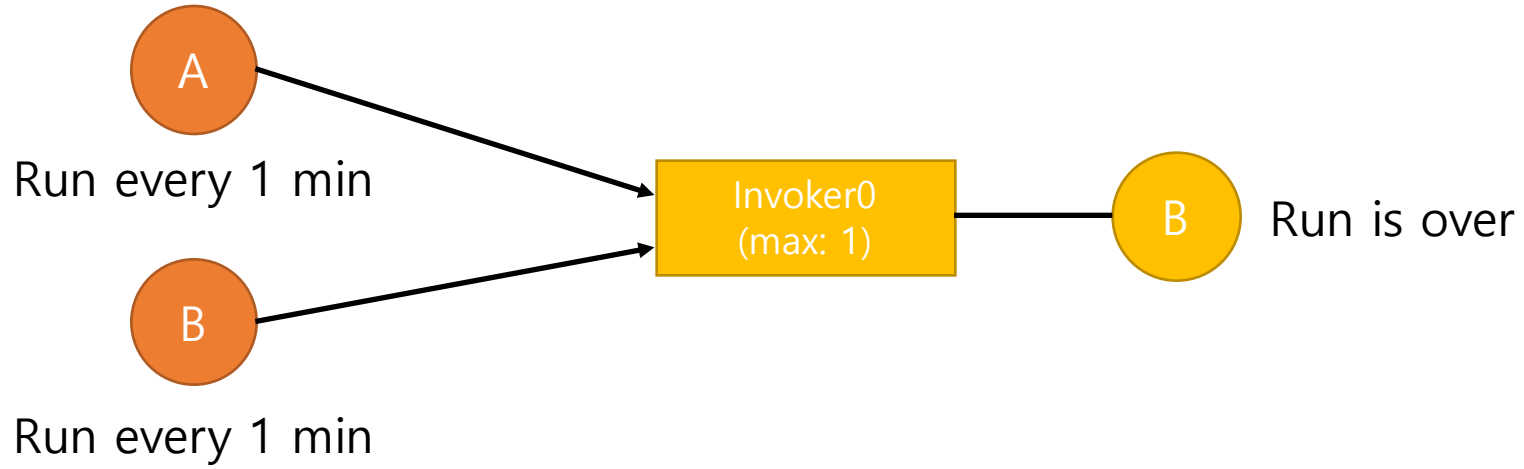
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



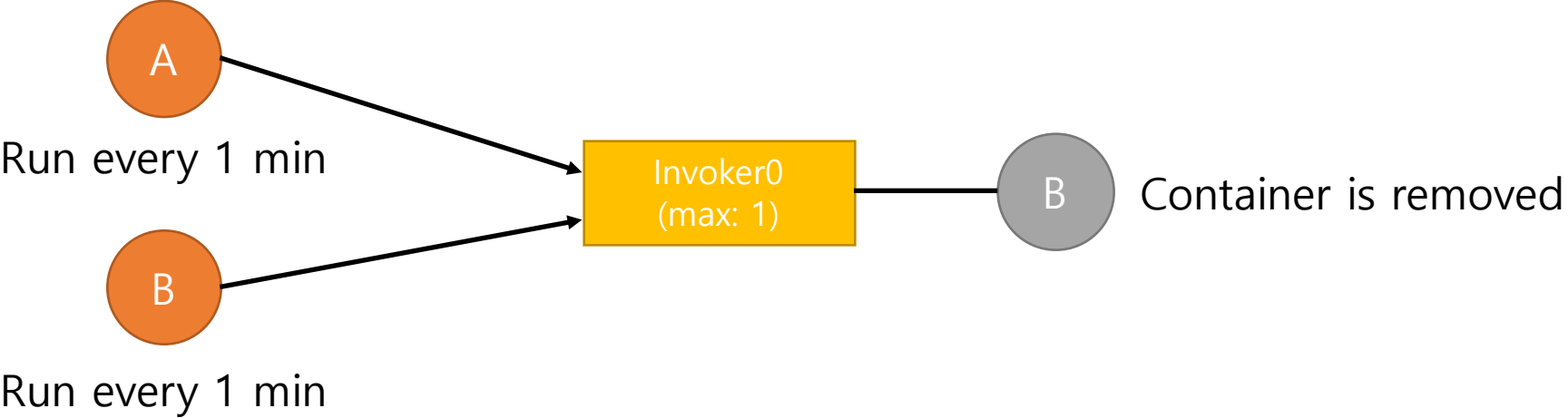
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



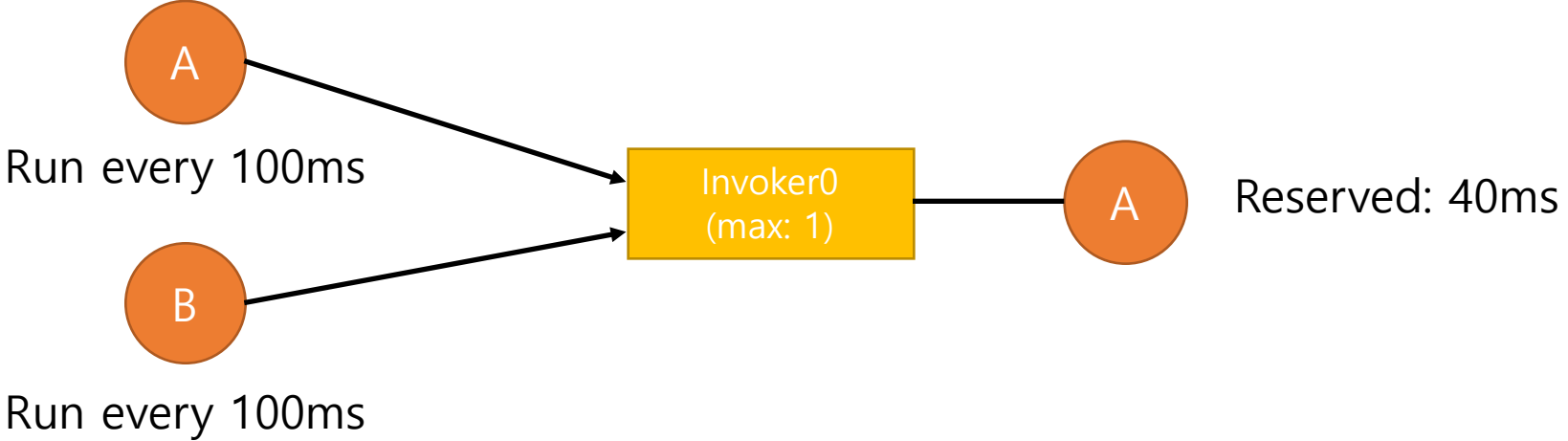
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



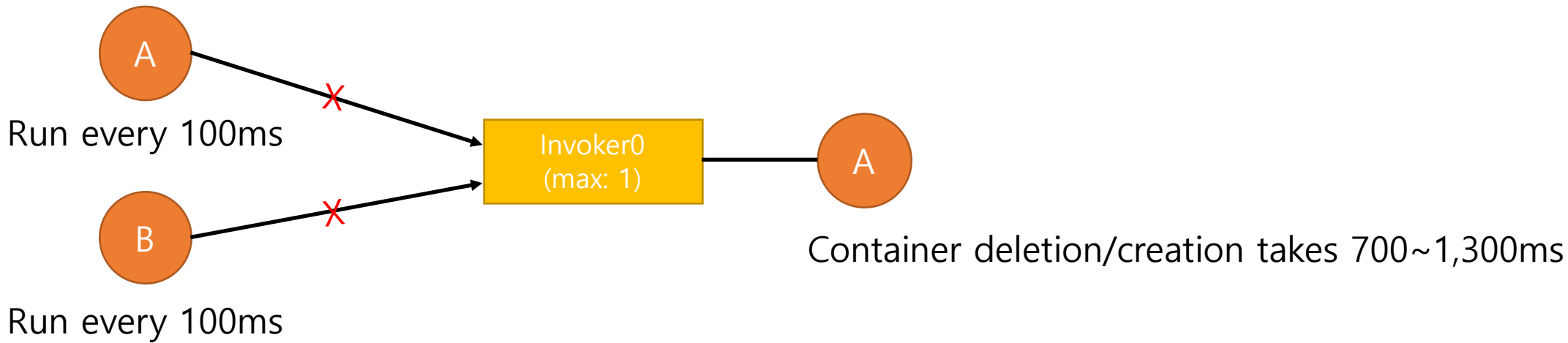
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



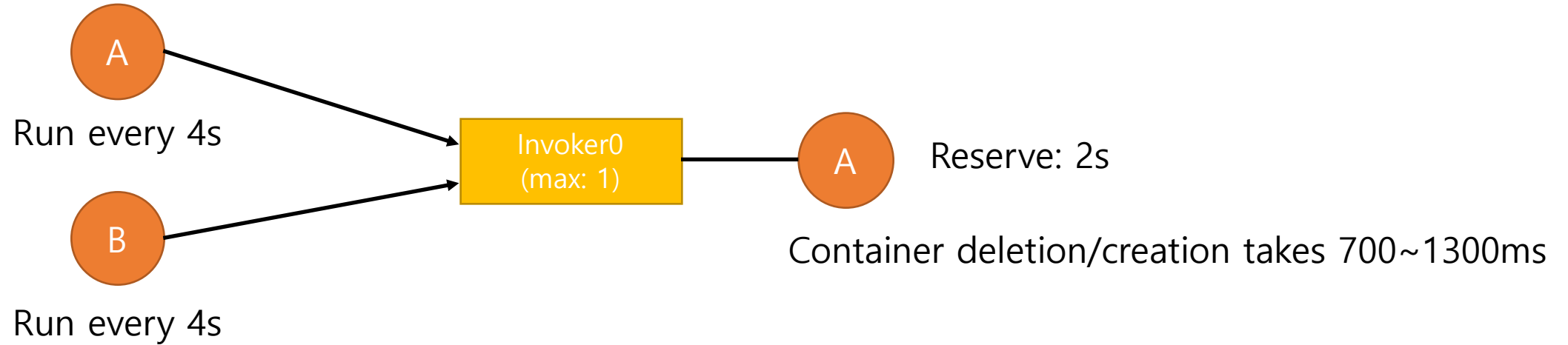
Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



Cons - Action container can be reserved for 30s ~ 1min (Overcommit)

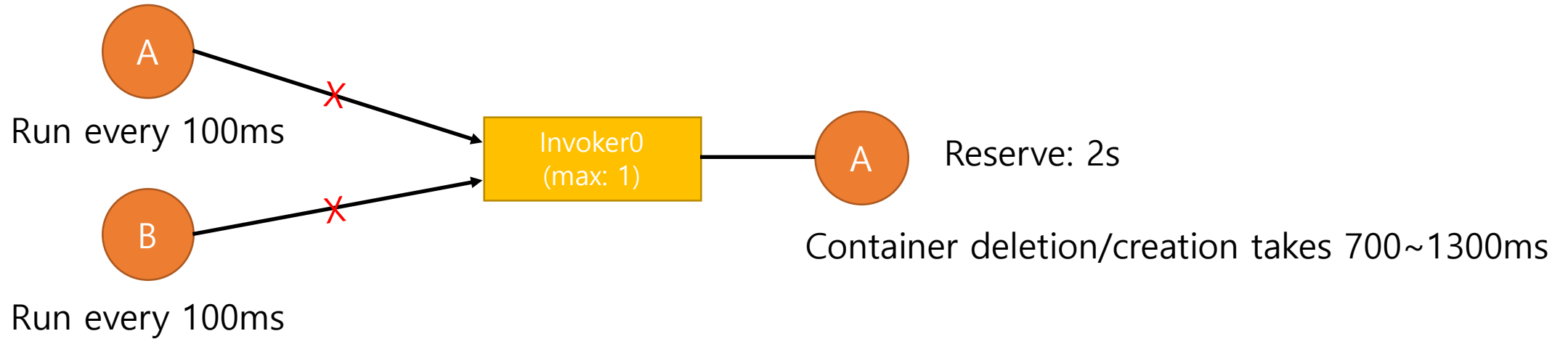


Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



Running interval can be changed.

Cons - Action container can be reserved for 30s ~ 1min (Overcommit)



Running interval can be changed.

If interval is slow enough and not occurred at the same time, all requests will be properly served. But if interval is lesser than 700~1,300ms, we cannot guarantee execution for multiple actions.

-> **Overcommit. Tradeoff: Resource utilization vs Performance guarantee**

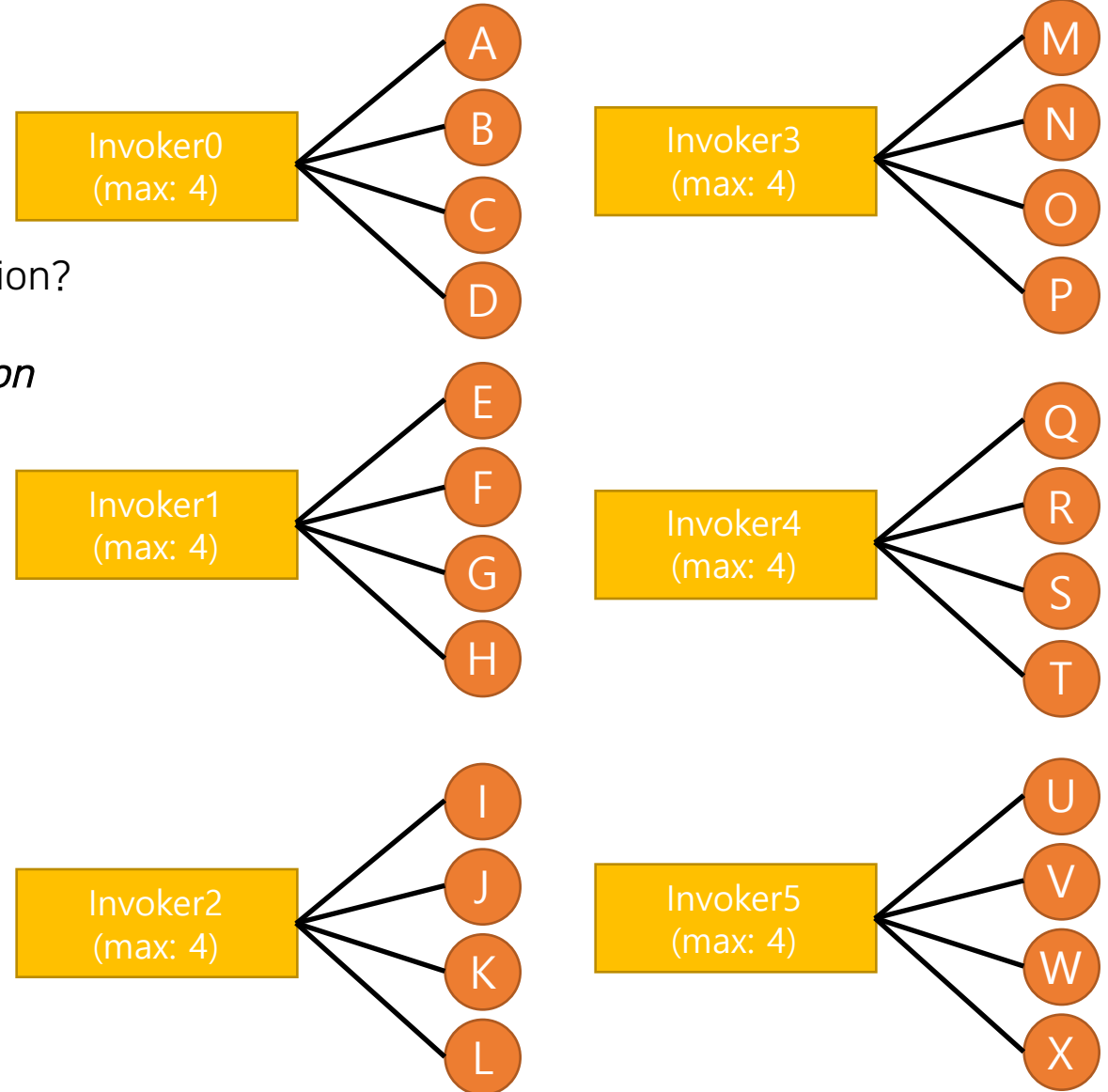
If we allow this, we can serve more actions with 1 container but can not guarantee performance.

Cons - Action container can be reserved for 30s ~ 1min (Overcommit)

Cluster is saturated at some point.

Which one is more proper approach under this situation?

1. *To control wait time to maximize resource utilization*
2. *To add more servers to guarantee performance*



Cons - Action container can be reserved for 30s ~ 1min (Overcommit)

Cluster is saturated at some point.

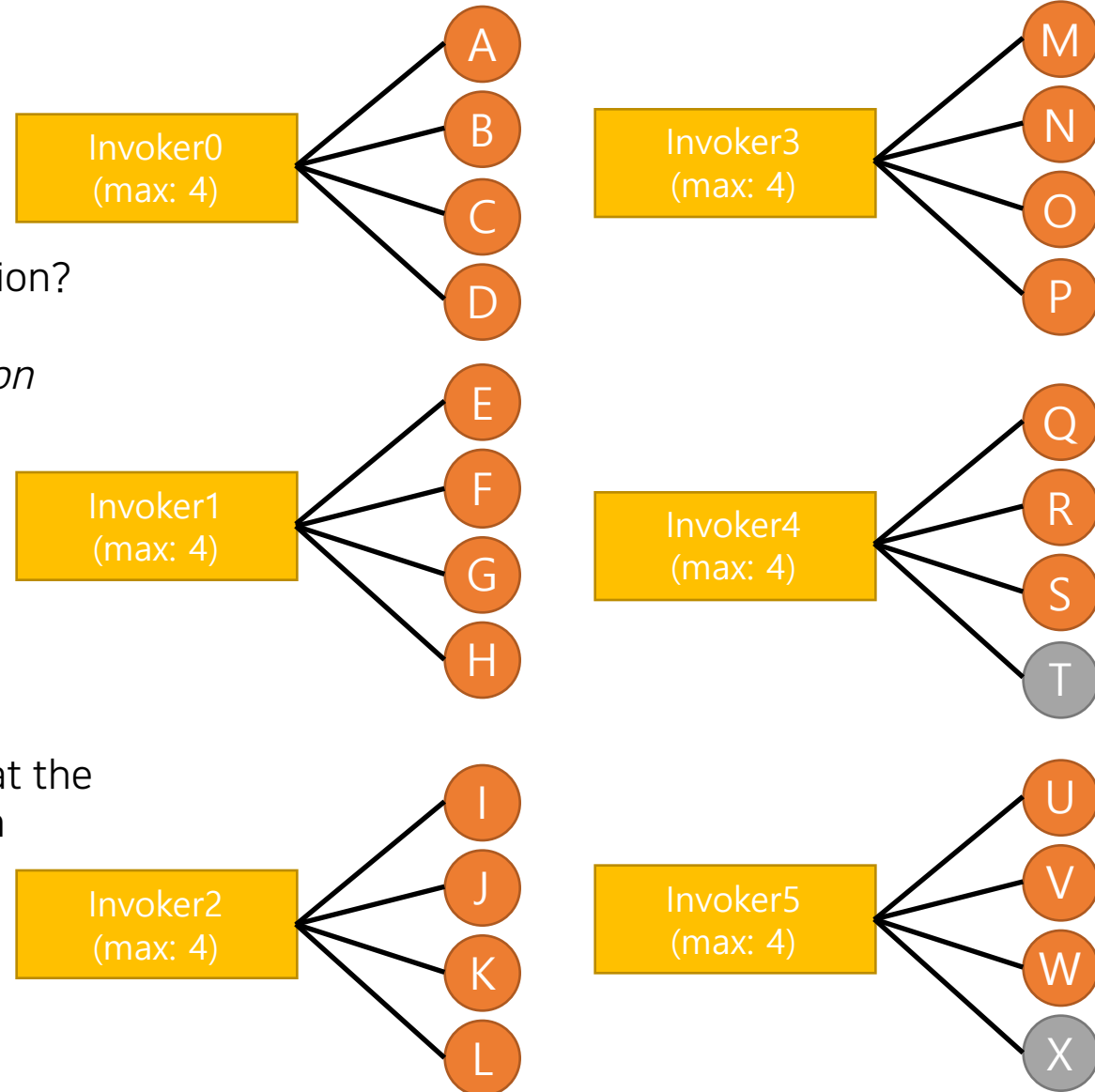
Which one is more proper approach under this situation?

1. To control wait time to maximize resource utilization
2. To add more servers to guarantee performance

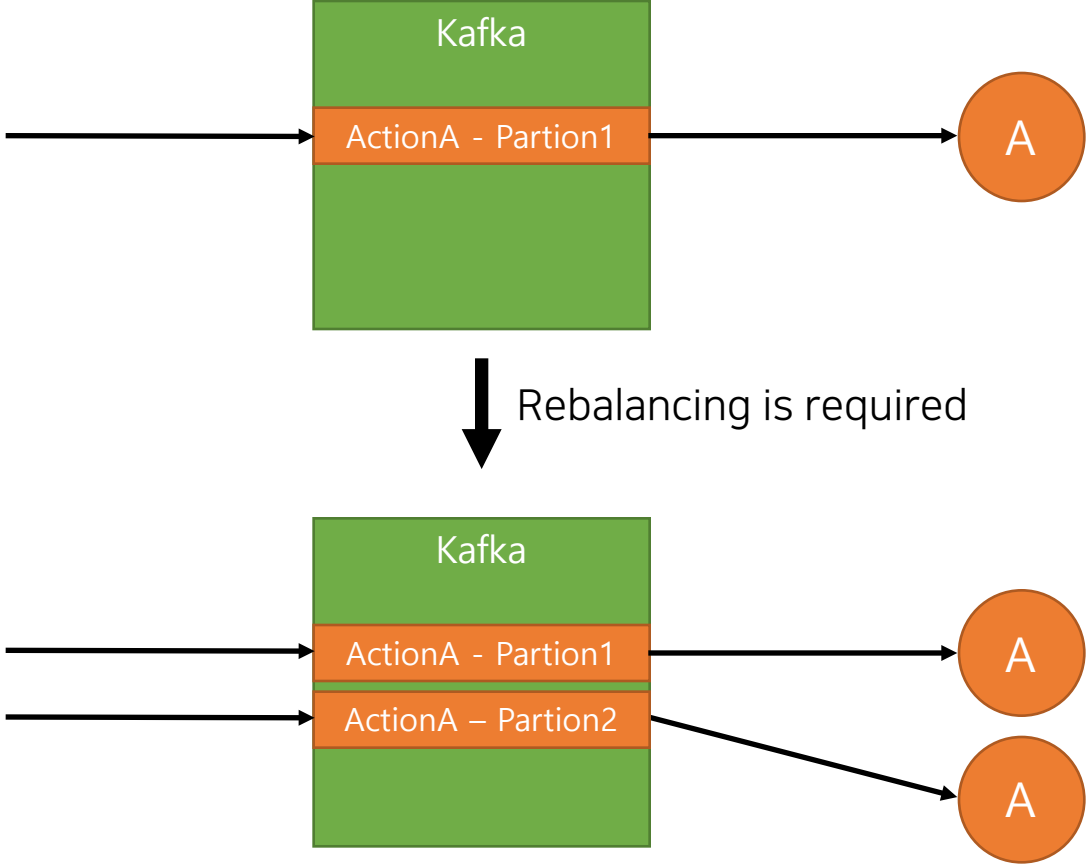
Performance guarantee is more important from the perspective of users. (consistent performance)

Furthermore, it only happens under the situation that the number of concurrent actions reaches the maximum number of concurrent containers.

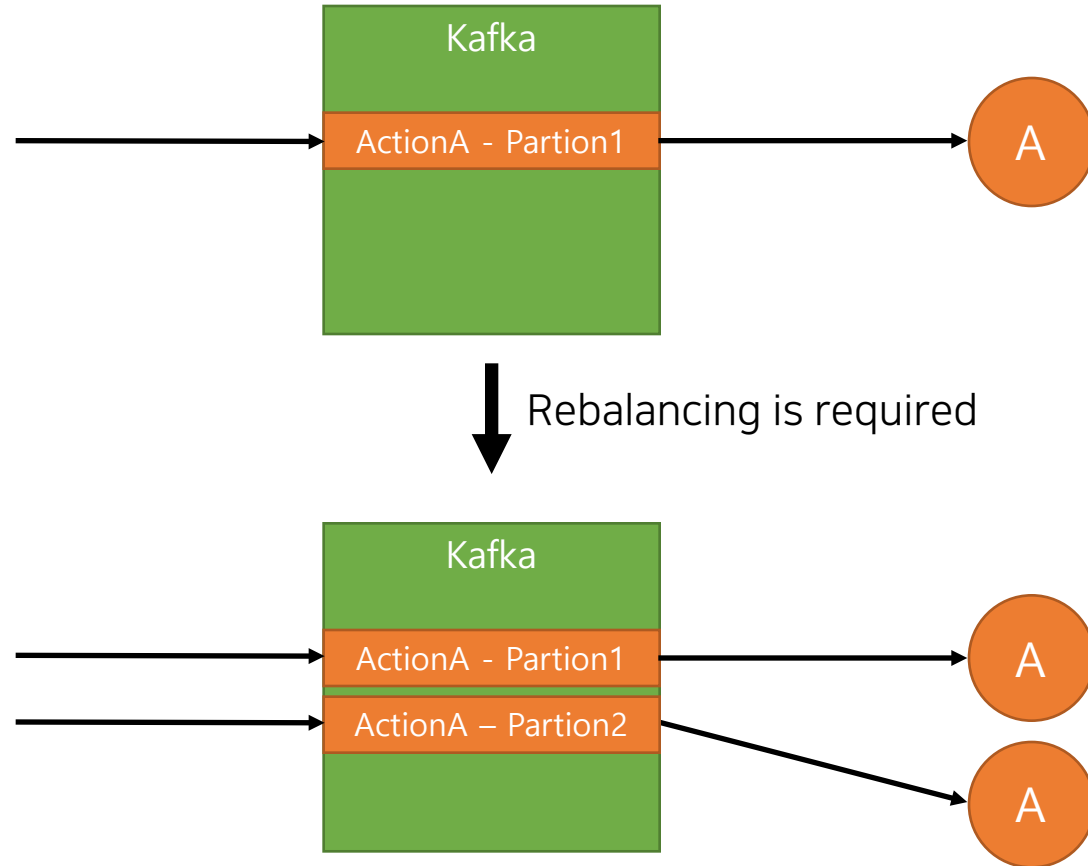
Anyway we need to find optimal reserving time.



Cons - If limit changed, # of partitions should be also changed.

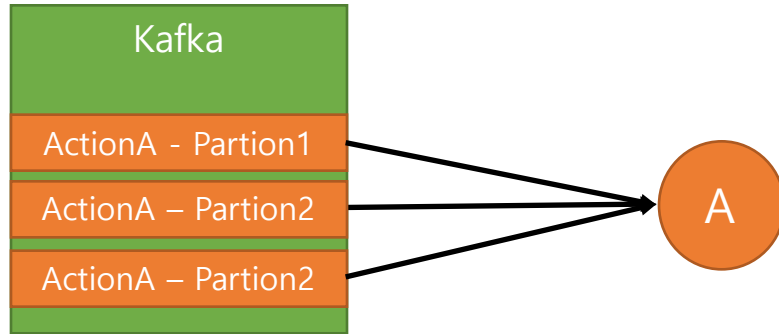


Cons - If limit changed, # of partitions should be also changed.



- We can limit retention(bytes, duration)
 - Once action is invoked, data is less meaningful.
 - Rebalancing takes not much time.
 - Lesser than 1s.
- Rebalancing only happens when changing limit.
 - Limit is not frequently changed

Cons - # of partitions cannot be decreased.

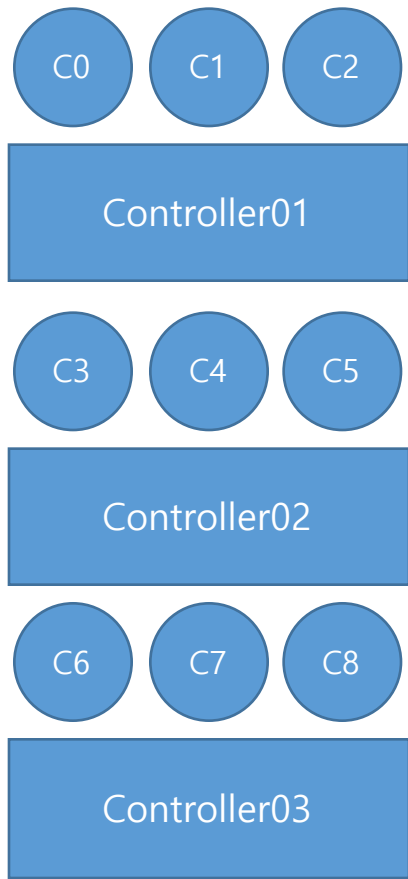


- Though there are multiple partitions, one consumer can read data from all of them
- Decreasing limit may not happen frequently.
- One option: We may explicitly delete and recreate topic when limit is decreased.

Performance Evaluation

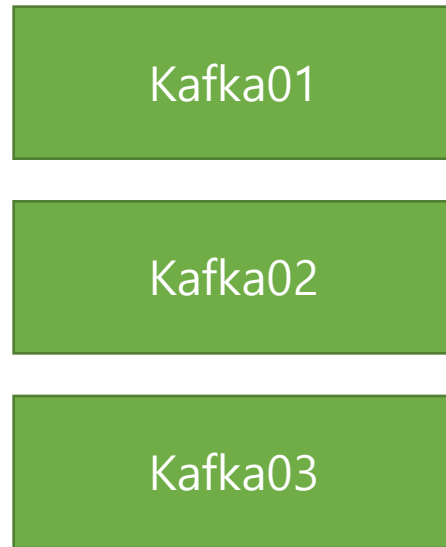
With prototype

Environment – BMT Environment



3 machines, 9 containers

8 cores, 16GB MEM, 100GB HDD



3 machines, 3 containers

40 cores, 128GB MEM, 2TB SSD



maxPoolSize: 20
Total Capa: 180
(1 for blackbox)

10machines, 10 containers

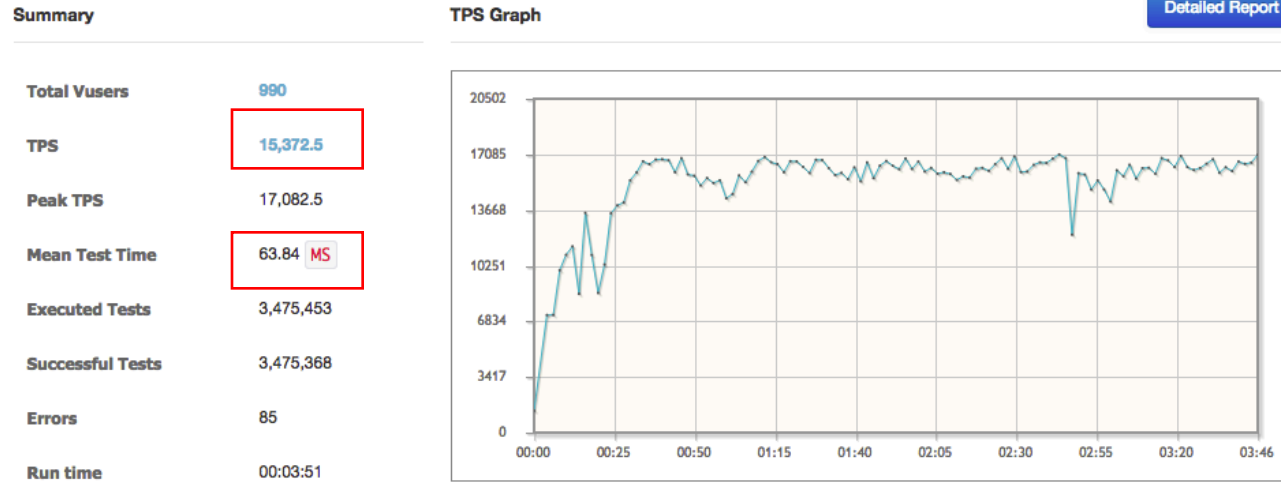
8 cores, 16GB MEM, 100GB HDD

Performance comparison - 1 action test

Current implementation

Description: 1 action with current implementation

Test Configuration | Report



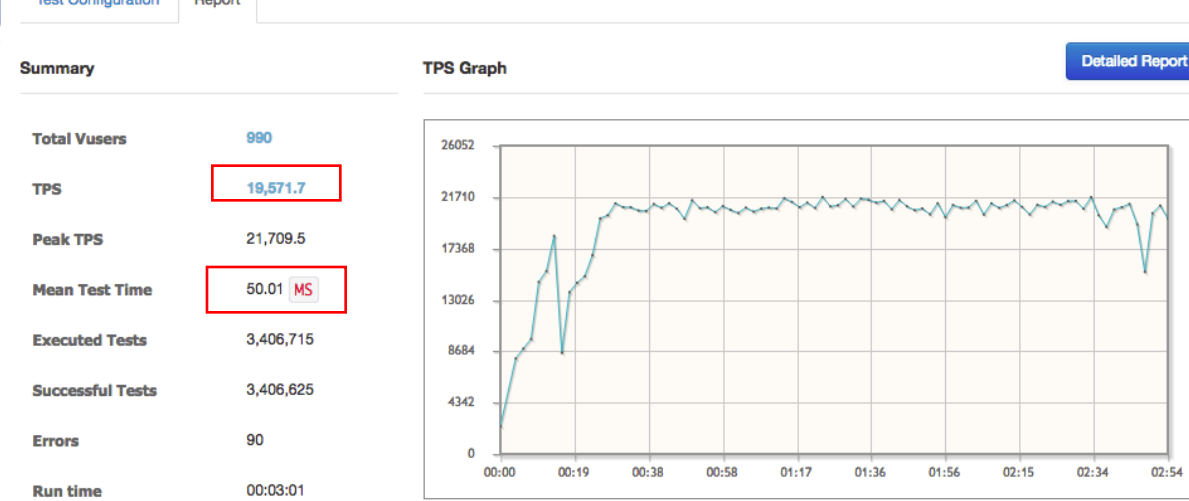
1 action with 180 containers

New implementation

Test Name: new scheduling algorithm | Tags: x_new_scheduling | Clone | Clone and Start

Description: 100 partition test in Self serving algorithm

Test Configuration | Report



1 action with 100 containers

1.33 times more TPS
1.26 times faster execution

Performance comparison - 100 actions test

Current implementation

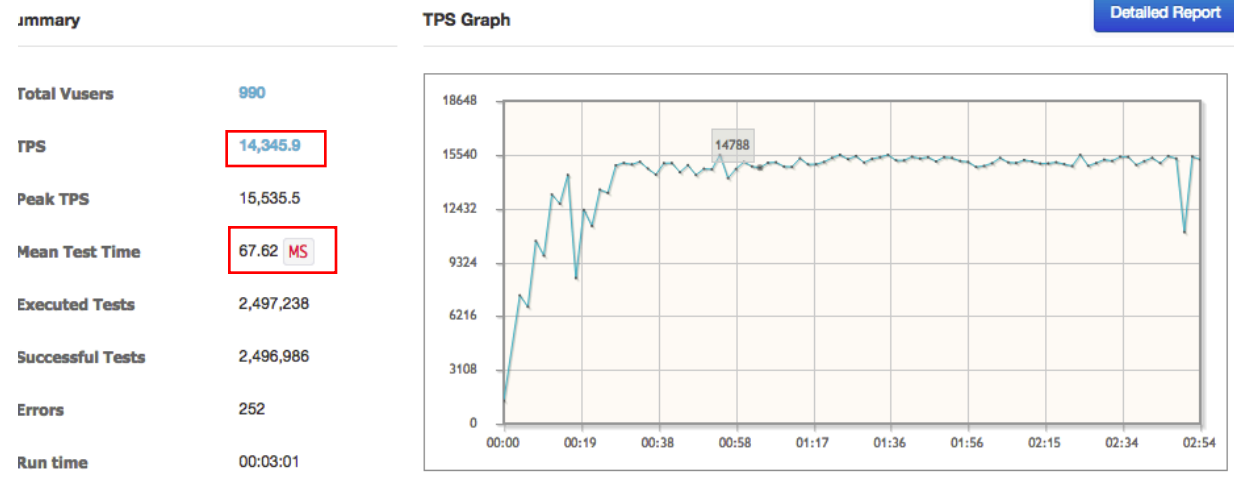
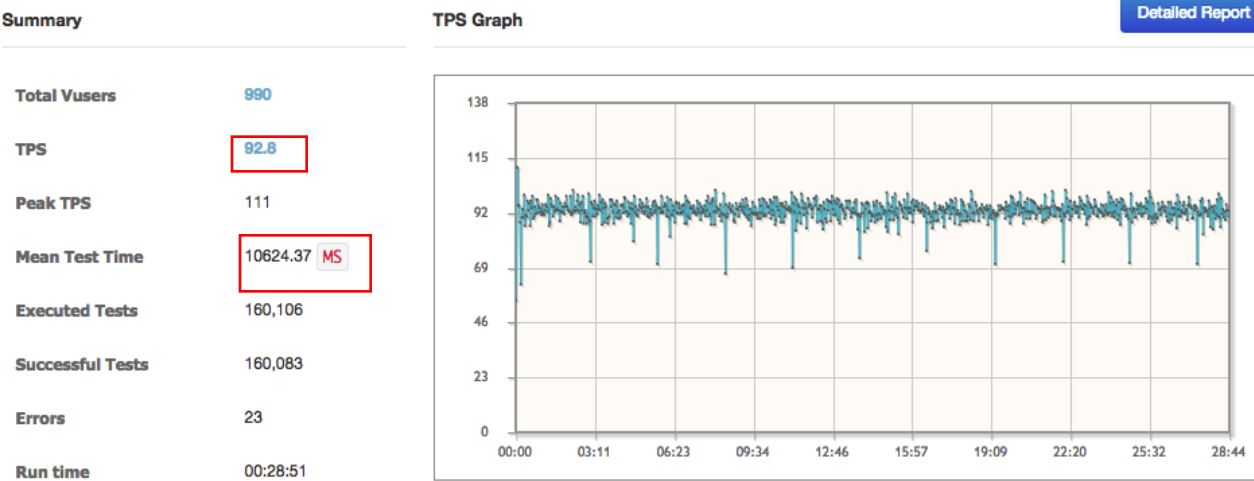
New implementation

Description 100 action with current implementation

[Test Configuration](#) [Report](#)

Description 100 actions with Self serving algorithm

[Test Configuration](#) [Report](#)



100 actions with 180 containers

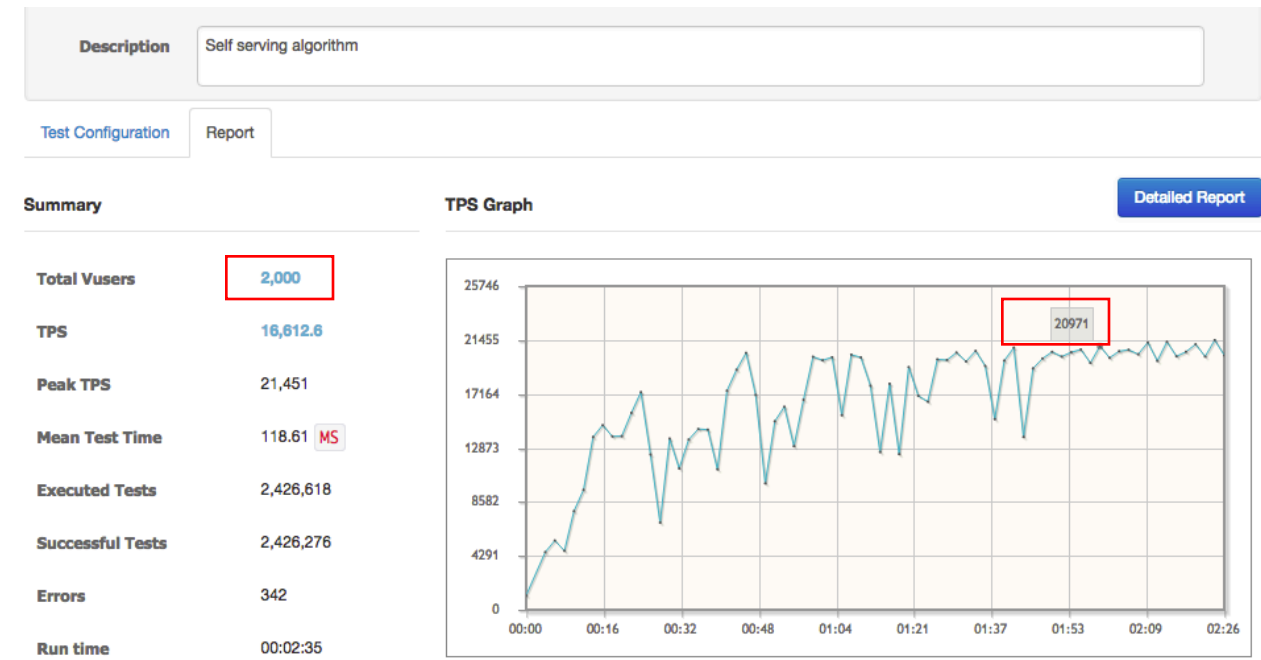
100 actions with 1 containers each(100 containers)

163 times more TPS
158 times faster execution

Performance comparison - 100 actions test with more loads

New implementation

New implementation



Just increased the number of Vusers
It showed similar performance with 1 action case
Even 80 containers were not utilized

Performance comparison – Long running tests(8 hours)

New implementation

Test Name **Tags** Clone Clone and Start

Description

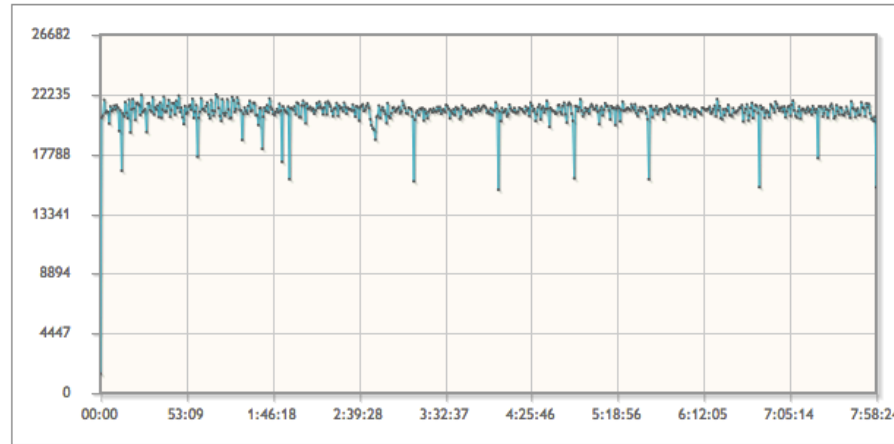
[Test Configuration](#) [Report](#)

Summary

TPS Graph

[Detailed Report](#)

Total Vusers	990
TPS	20,955.8
Peak TPS	22,376
Mean Test Time	44.52 MS
Executed Tests	602,374,006
Successful Tests	602,163,850
Errors	210,156
Run time	07:59:00



100 actions with 1 containers each(100 containers)

Steady performance