데이터사이언스를위한 지식관리시스템

7주차: Kafka /MQTT cluster (with NiFi)

7주차 목표

- 6주차에서 했던 내용들을 Apache NiFi 환경에서 모두 수행해봅니다
- Kafka topic의 partition을 분할해서 분산처리를 수행해봅니다
- Kafka topic의 분산처리시 주의해야 할 사항에 대해 알아봅니다

Apache NiFi

Apache NiFi

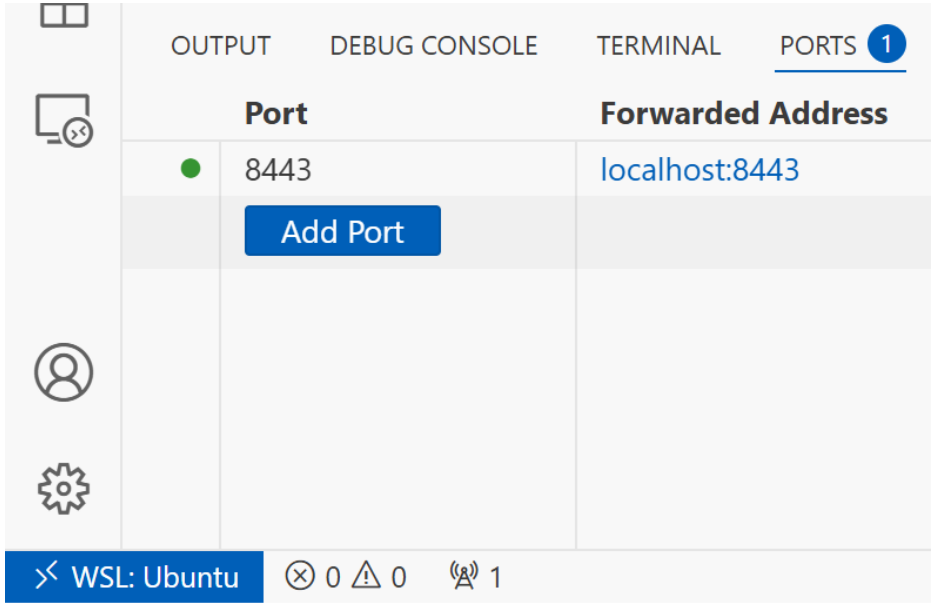
- Niagara Files
- 폭포처럼 흐르는 데이터

- NiFi2: NiFi에서는 강력히 권장하는 버전, 그러나 JDK21이 필요하며 ip접속시 추가 설정 필요
- NiFi1: 비교적 최신 시점에 마지막 Minor 버전 업데이트(2024-12) JDK11에서 사용 가능

- NiFi1으로 선택하여 아래 문서를 참고하여 설치합니다
- https://github.com/apache/nifi/tree/rel/nifi-1.28.1

Apache NiFi (Windows -> WSL/SSH)

- WSL도 일종의 가상머신 입니다
- WSL의 ip에 접근하거나 port forwarding 설정을 해야 합니다
- 좀더 편한 방법으로 port forwarding을 사용합니다
- Terminal을 띄운 후 그 옆에 있는 PORTS탭에서 8443 port를 추가합니다



Apache NiFi 실행

- JDK11이 설치된 이후에 아래와 같이 설치하고 실행할 수 있습니다
- 이 때 계정 정보가 필요합니다

```
wget https://dlcdn.apache.org/nifi/1.28.1/nifi-1.28.1-bin.zip
unzip nifi-1.28.1-bin.zip
cd nifi-1.28.1-bin
./bin/nifi.sh start
```

- https://github.com/apache/nifi/tree/rel/nifi-1.28.1 의 Authenticating 을 참고합니다

```
cat logs/nifi-app* | grep Generated
```

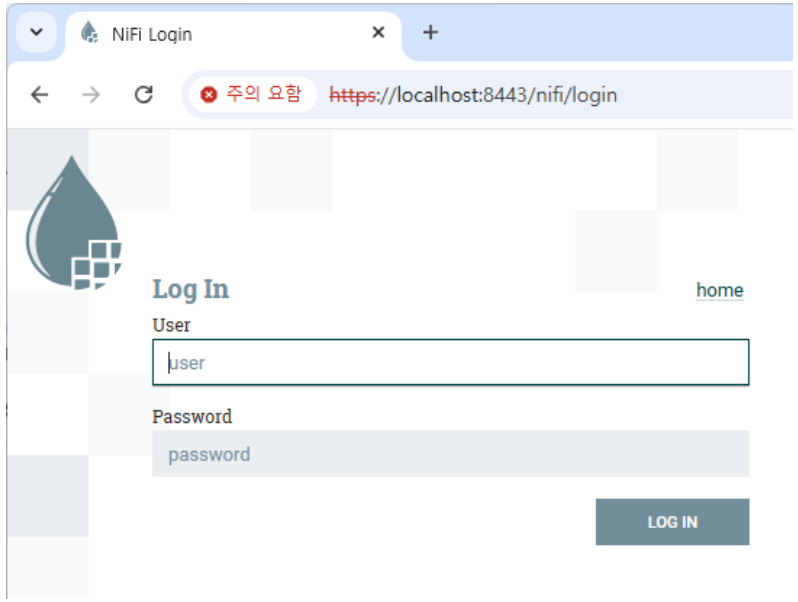
- 아래 [ ] 안에 있는 내용이 username/password 입니다

```
Generated Username [960e5b13-6f26-487e-9906-f621aaa50f29]
Generated Password [h8Ws6aitRxL4U8slAoAfES6ob/n8DGtv]
```

Apache NiFi의 WebUI 접속

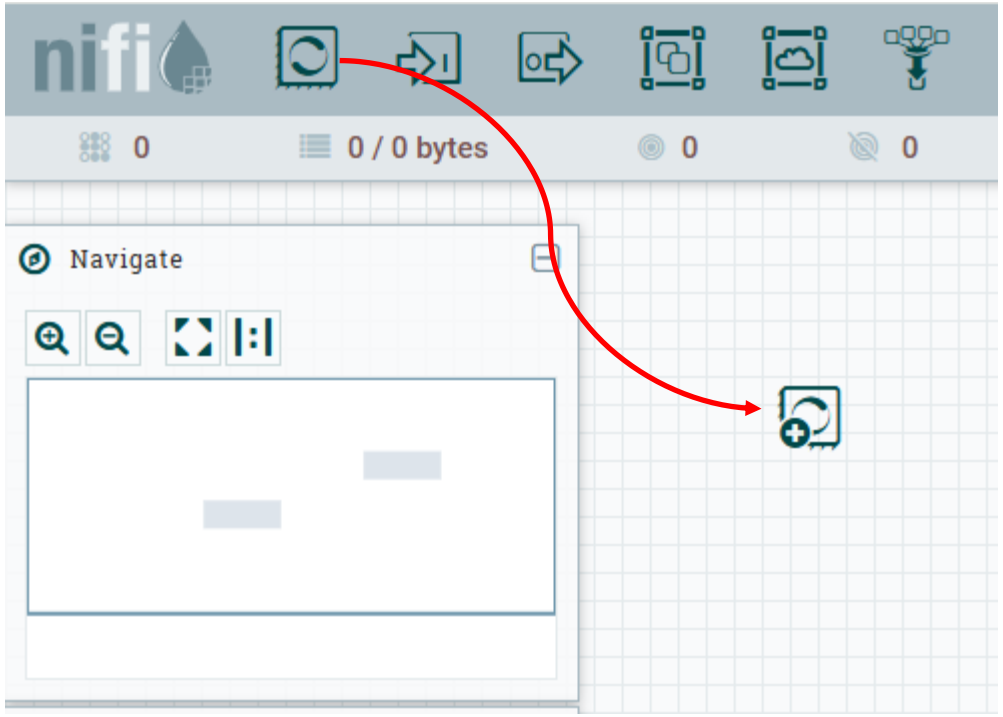
- WSL(ssh) + Port forwarding을 사용하셨다면 아래와 같이 진행합니다
- https://localhost:8443/nifi
- 여기서 localhost는 분명히 Windows host가 맞습니다
- 그런데 Port forwarding이 8443이 원격지(WSL 또는 SSH)의 8443으로 연결을 해줍니다
- 이 때 username/password는 이전에 확인한 정보를 사용합니다

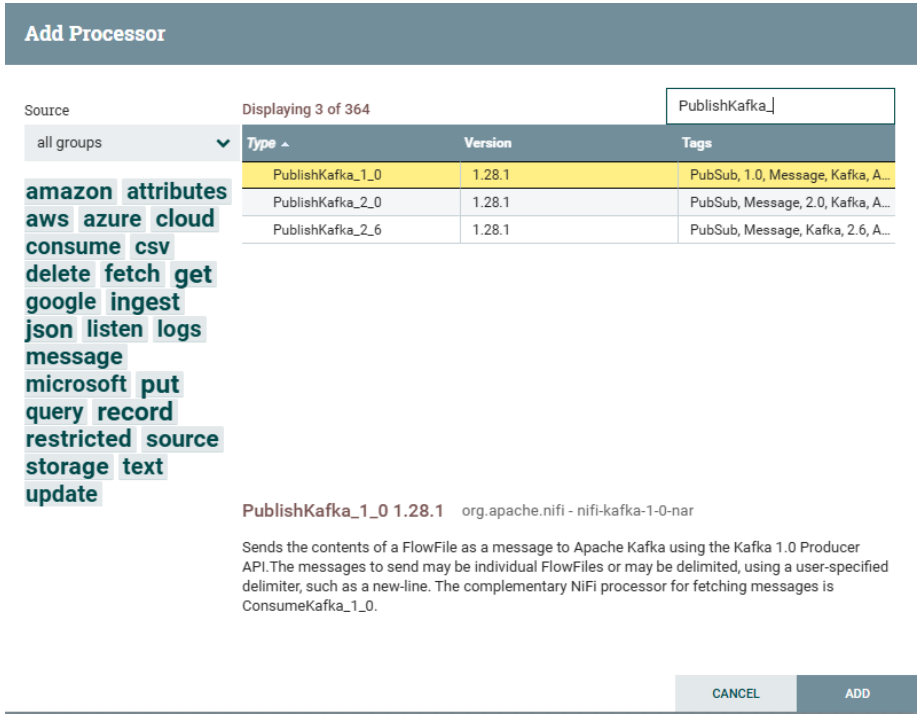
- 아래와 같은 Flow로 구성해보겠습니다
- File -> Producer -> Consumer -> File



Apache NiFi의 KafkaProducer

- 아래와 같이 드래그앤 드롭으로 Processor를 추가합니다
- PublishKafka_ 로 검색하고 PublishKafka_2.0을 선택하고 ADD 버튼을 눌러 완료합니다

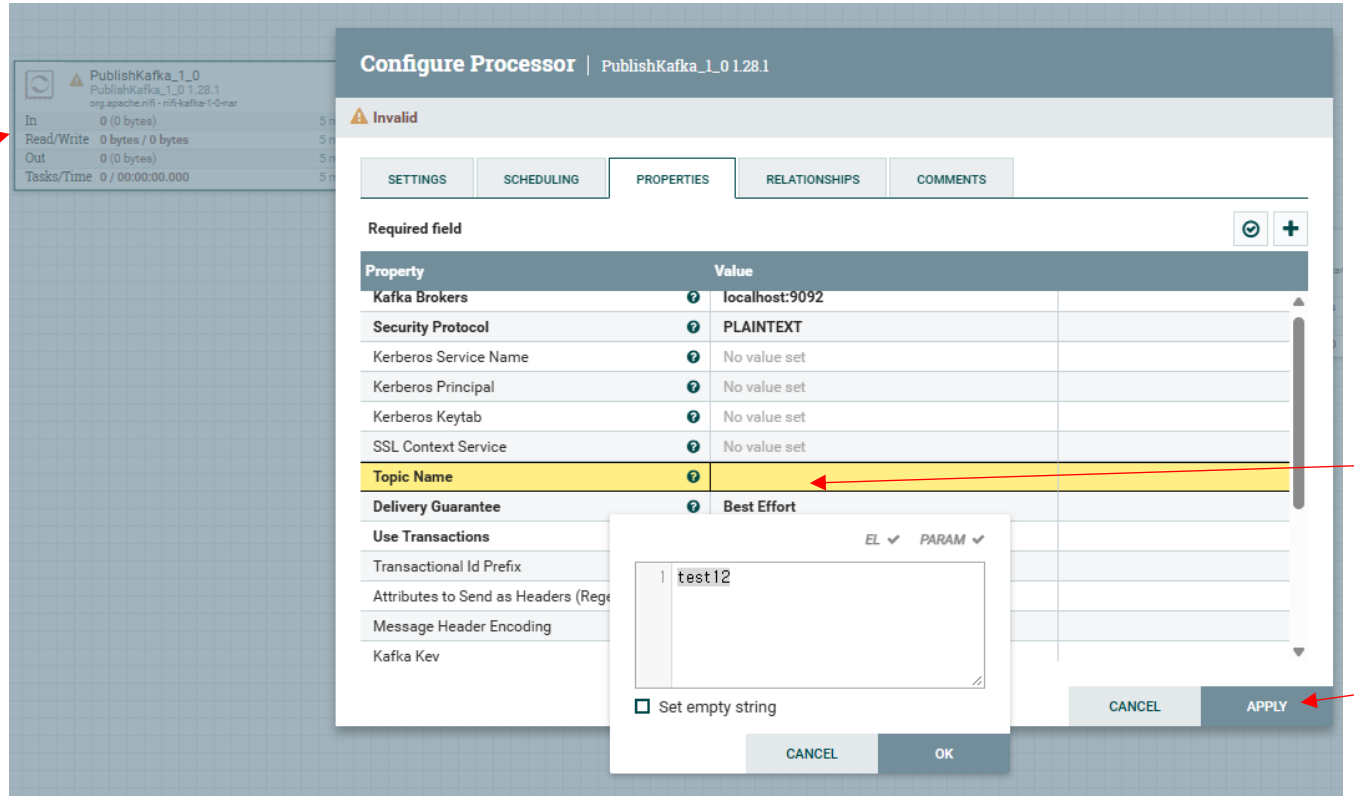




Apache NiFi의 KafkaProducer

- 추가된 Processor를 다시 더블클릭하면 Topic Name을 설정할 수 있습니다

더블클릭

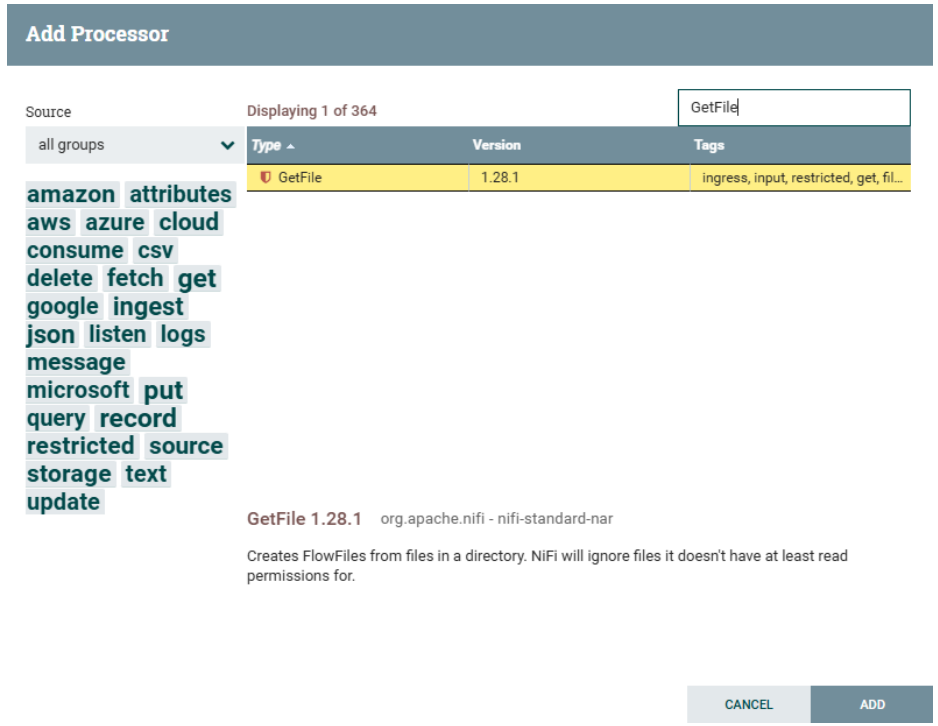


Topic 설정

Apply

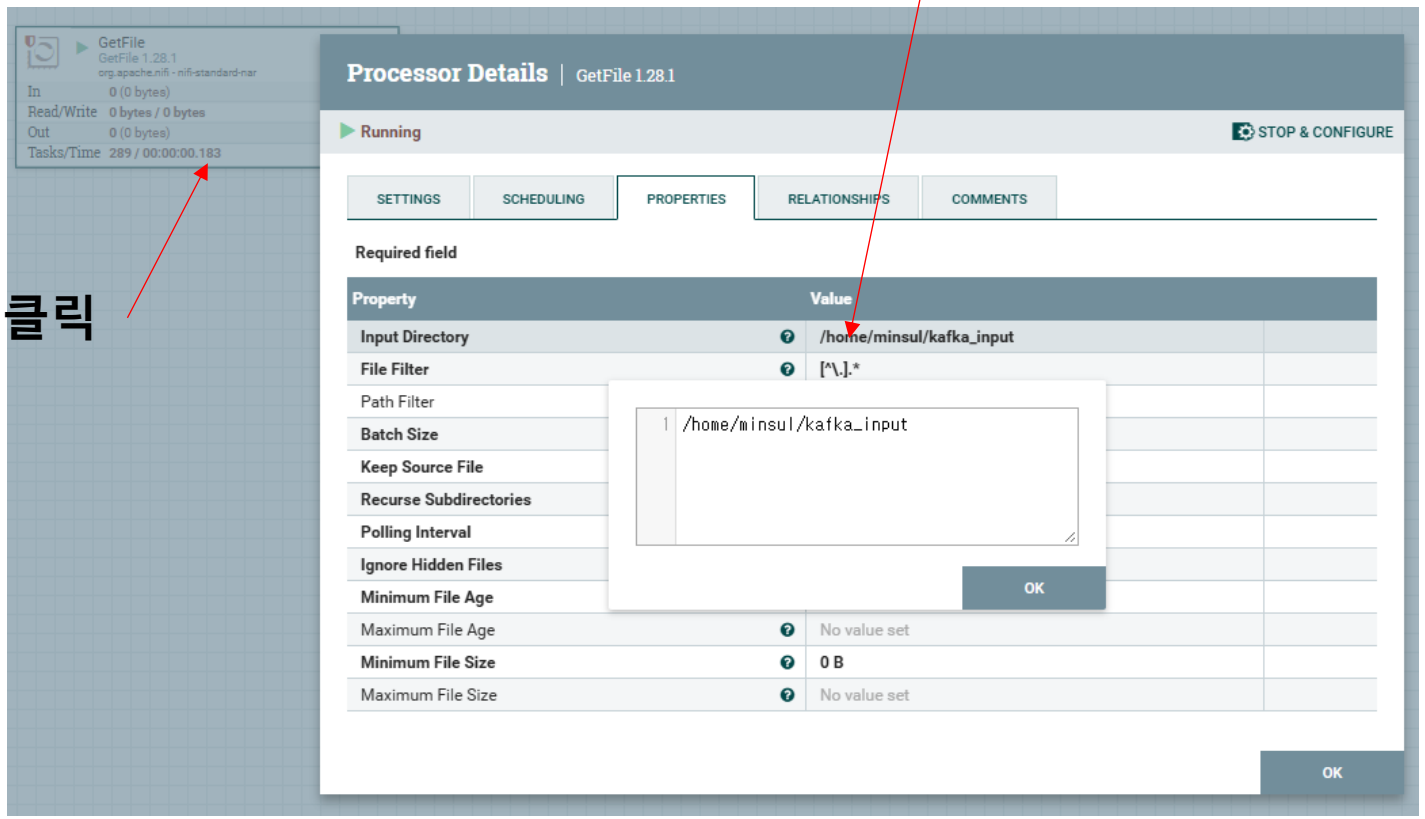
KafkaProducer의 데이터 소스 -> GetFile

- Processor를 새로 추가하고 GetFile을 선택합니다
- 만들어진 Processor를 다시 더블클릭하고 Input Directory를 지정합니다



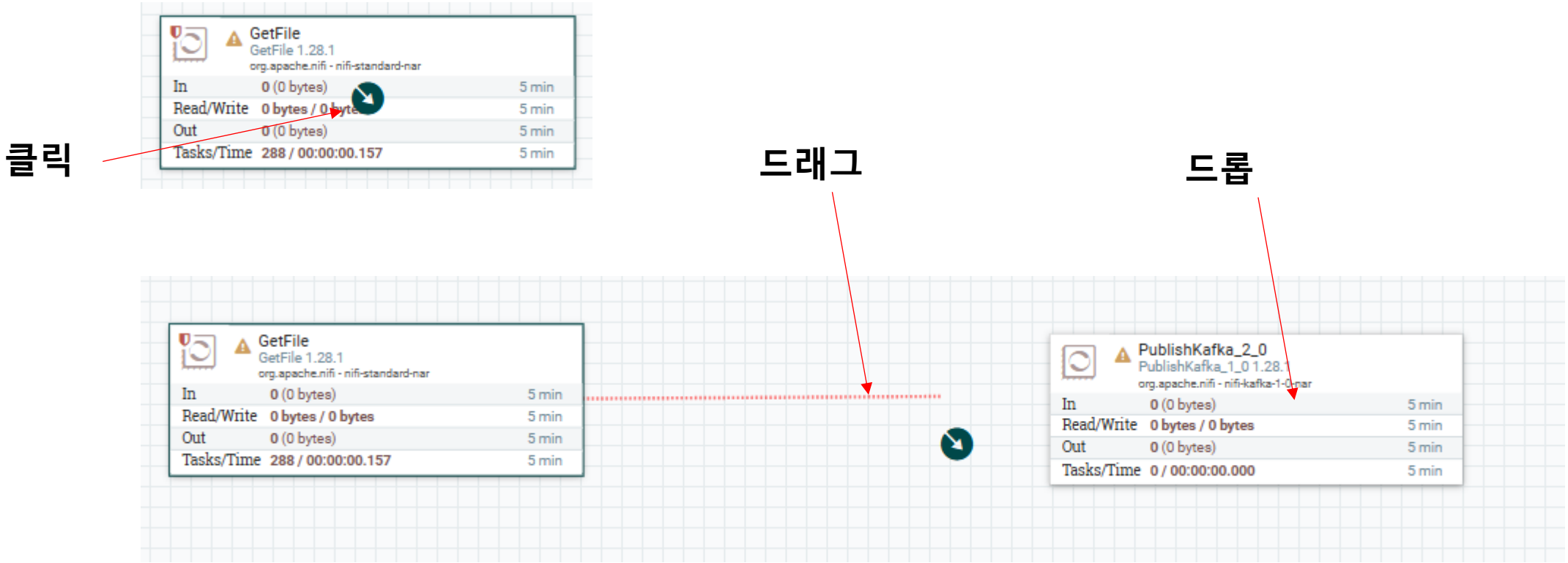
Input Directory 설정

더블클릭



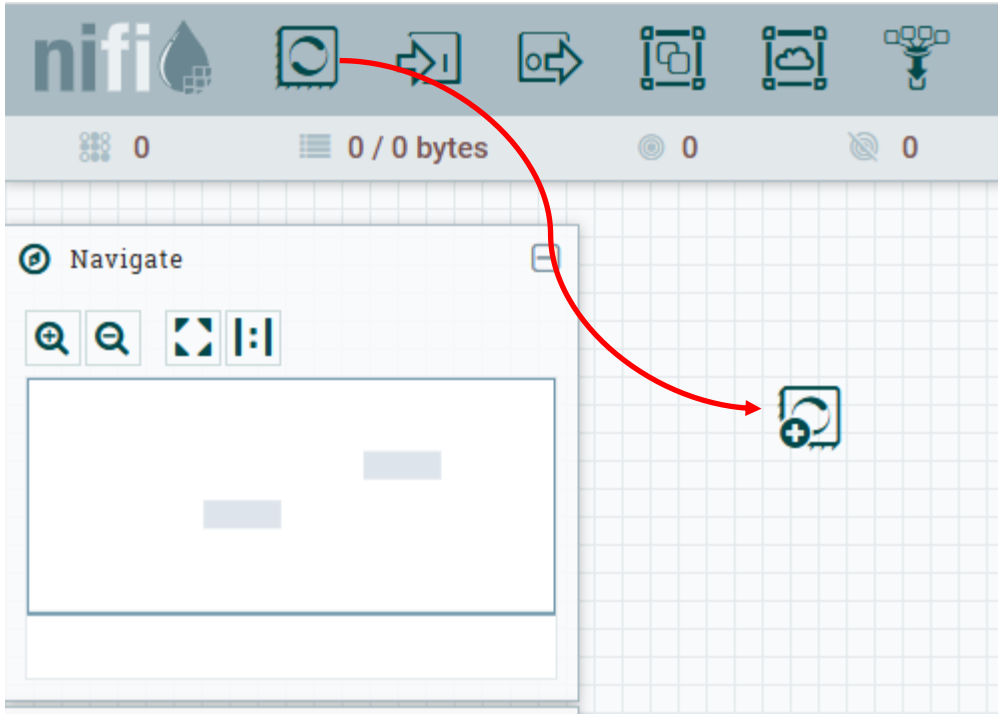
GetFile -> PublishKafka 연결

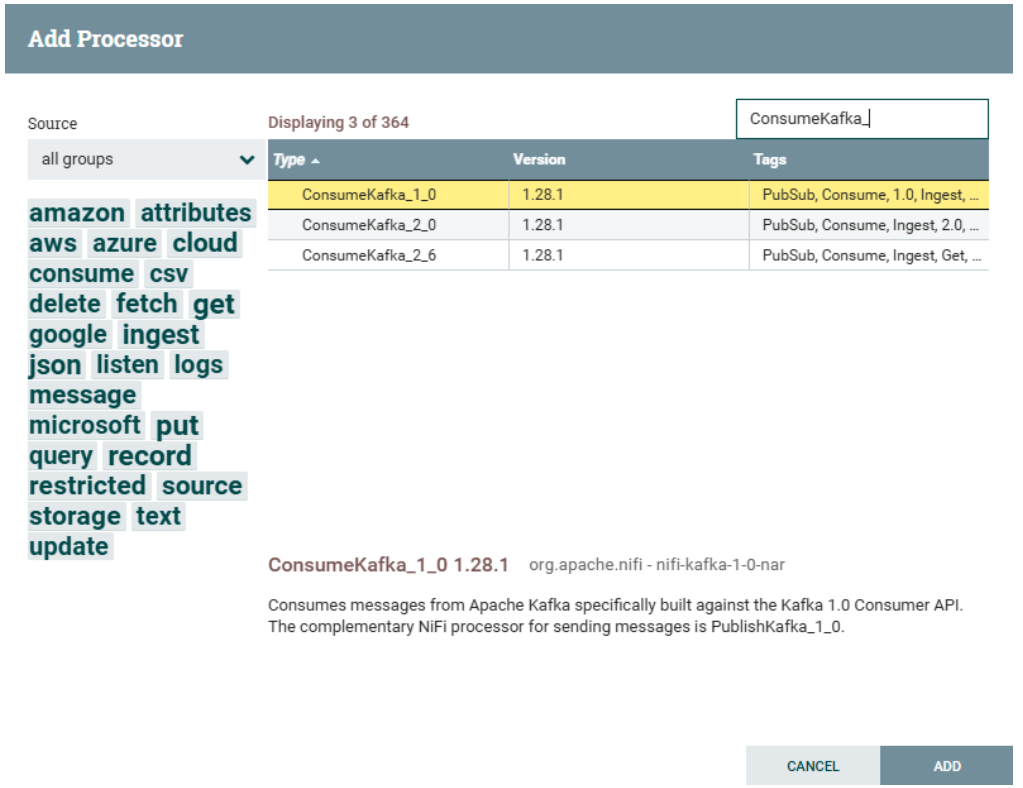
- 아래와 같이 GetFile에 있는 화살표를 클릭하고 드래그하여 PublishKafka에서 드롭합니다



Apache NiFi의 KafkaConsumer

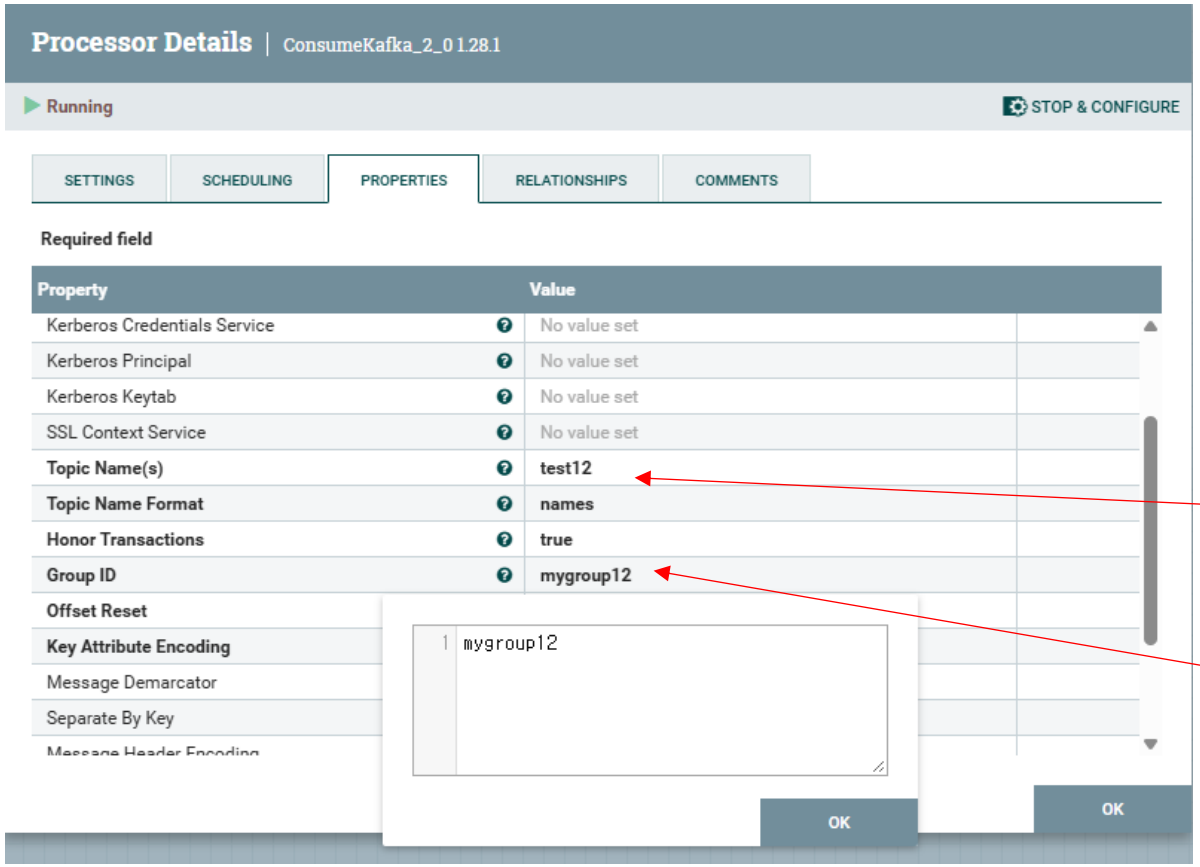
- 아래와 같이 드래그앤 드롭으로 Processor를 추가합니다
- ConsumeKafka_ 로 검색하고 ConsumeKafka_2.0을 선택하고 ADD 버튼을 눌러 완료합니다





Apache NiFi의 KafkaConsumer

- 추가된 Processor를 다시 더블클릭하면 Topic Name을 설정할 수 있습니다
- Consumer의 경우 group id도 설정해주어야 합니다



Topic 설정

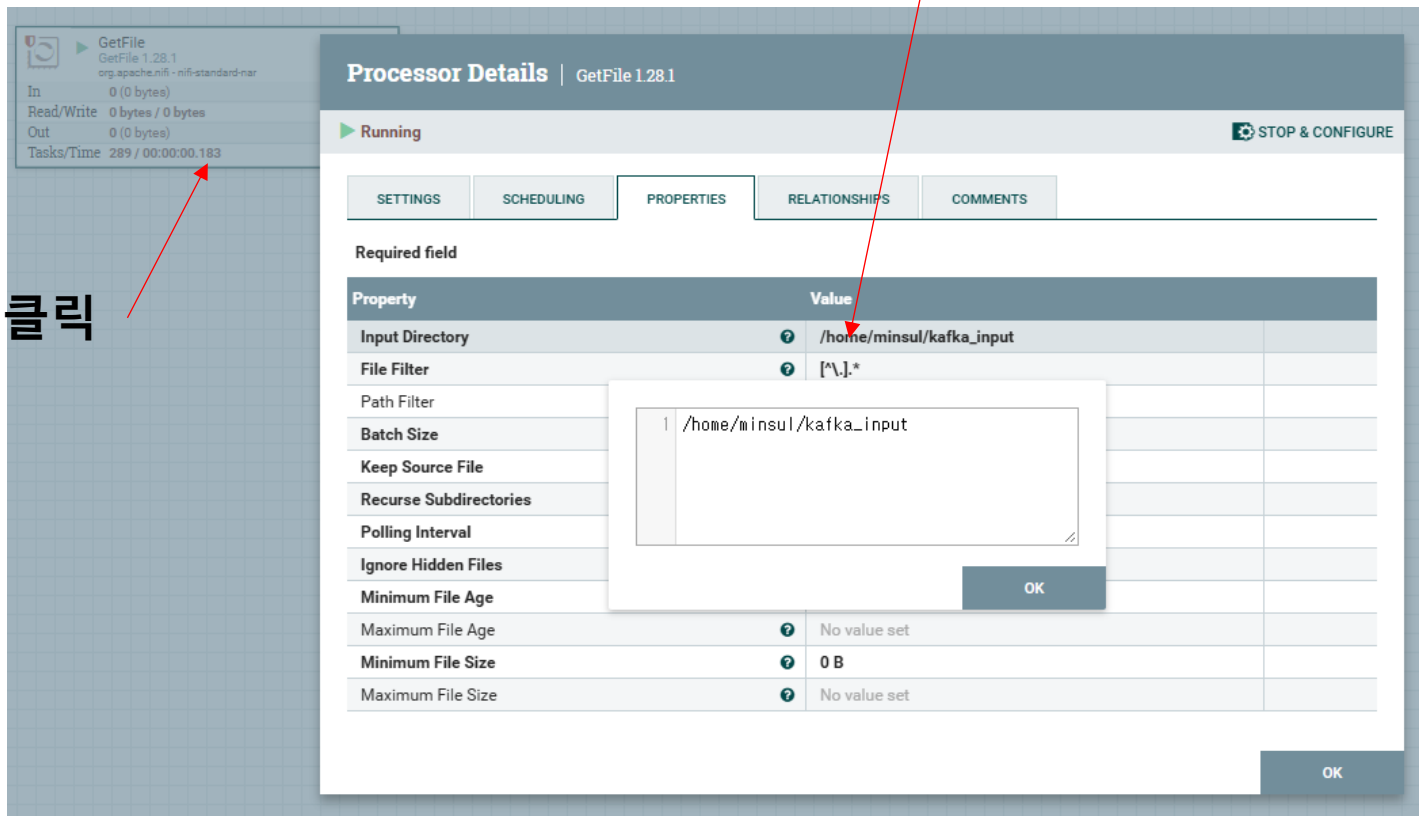
Consumer group ID 설정

KafkaConsumer의 타겟 -> PurFile

- Processor를 새로 추가하고 GetFile을 선택합니다
- 만들어진 Processor를 다시 더블클릭하고 Input Directory를 지정합니다

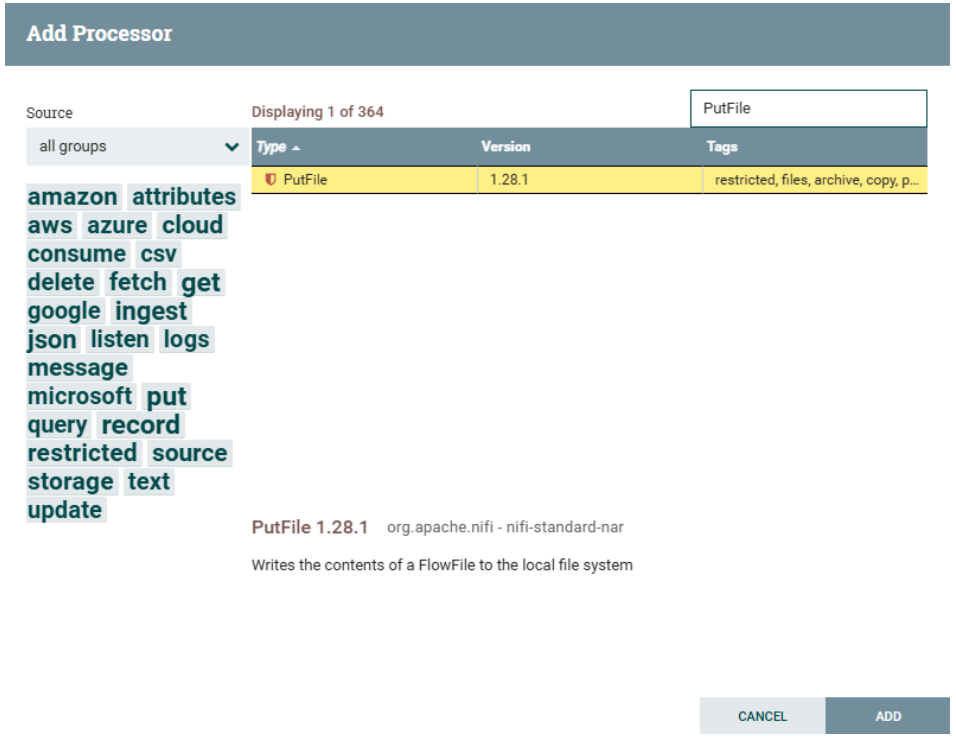
Input Directory 설정

더블클릭

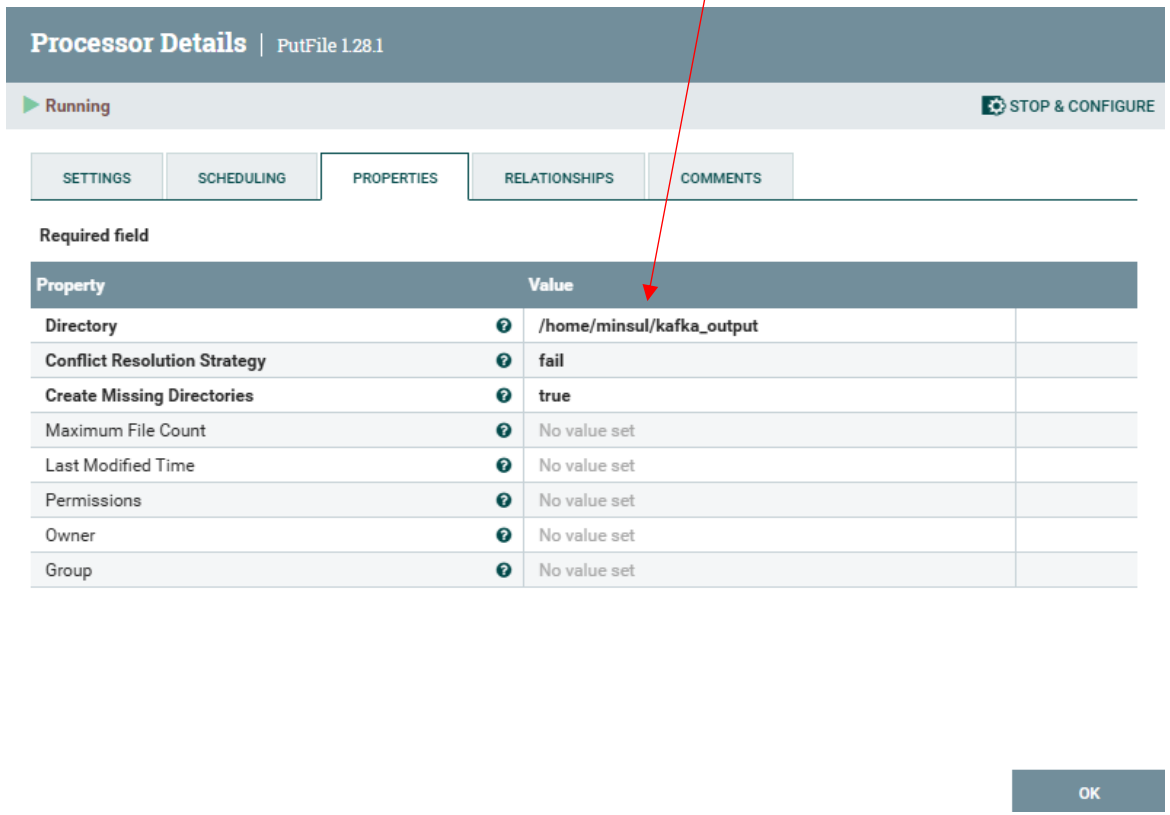


KafkaProducer의 데이터 소스 -> GetFile

- Processor를 새로 추가하고 PutFile을 선택합니다
- 만들어진 Processor를 다시 더블클릭하고 Directory를 지정합니다

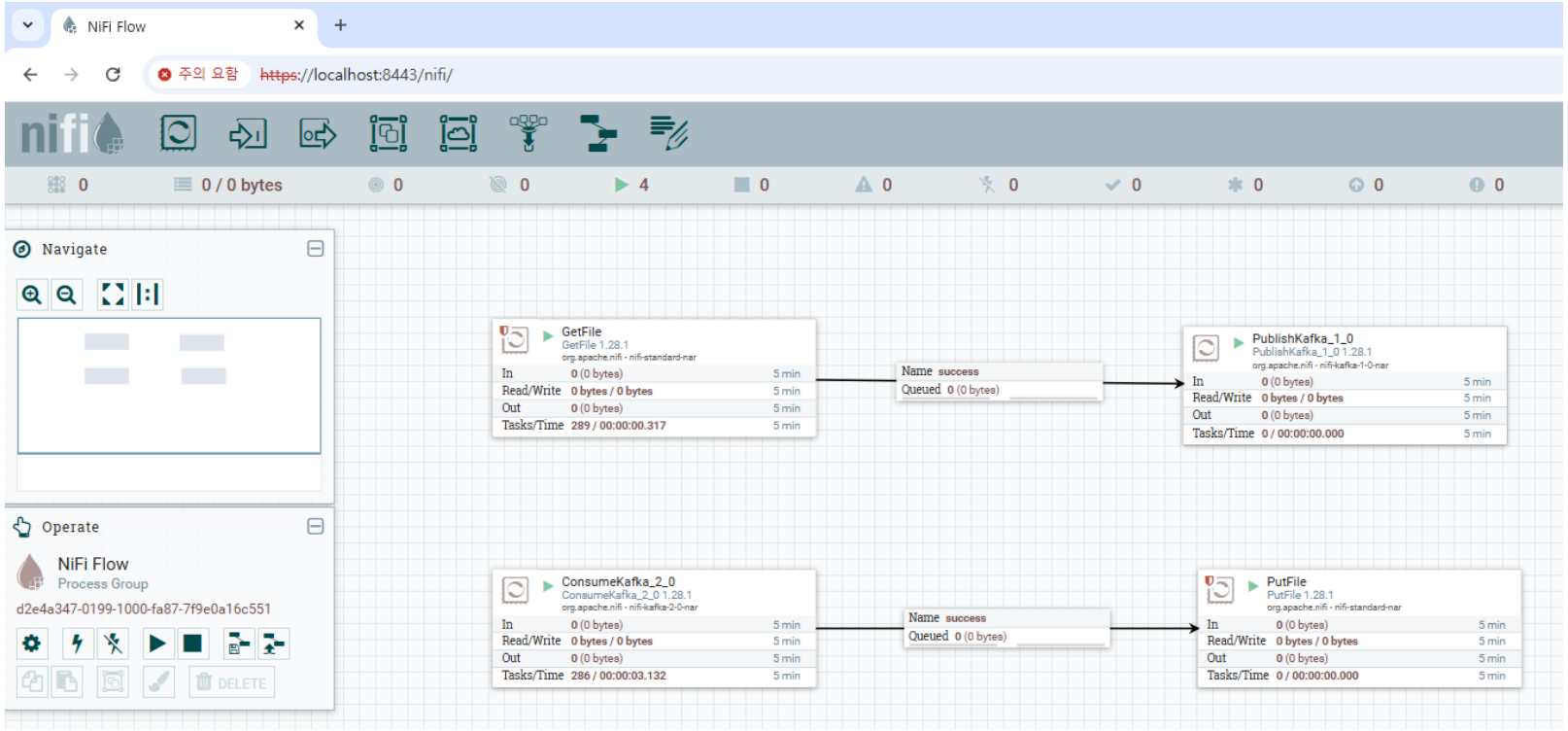


output Directory 설정



완성된 Kafka Pub/Sub 구조

- GetFile -> PubKafka -> ConsumeKafka -> PutFile 구조입니다
- GetFile에서 설정한 디렉토리에 임의의 텍스트파일을 만들어봅니다
- 이후 PutFile에 텍스트파일이 생기는 것을 확인할 수 있습니다



정리

- Apache NiFi를 이용하여 Kafka Producer/Consumer를 구성해보았습니다
- Apache Kafka 자체는 UI를 제공하지 않기 때문에 이러한 보조 도구를 이용하면 데이터의 흐름을 시각적으로 확인하기에 편리합니다

Kafka cluster

개요

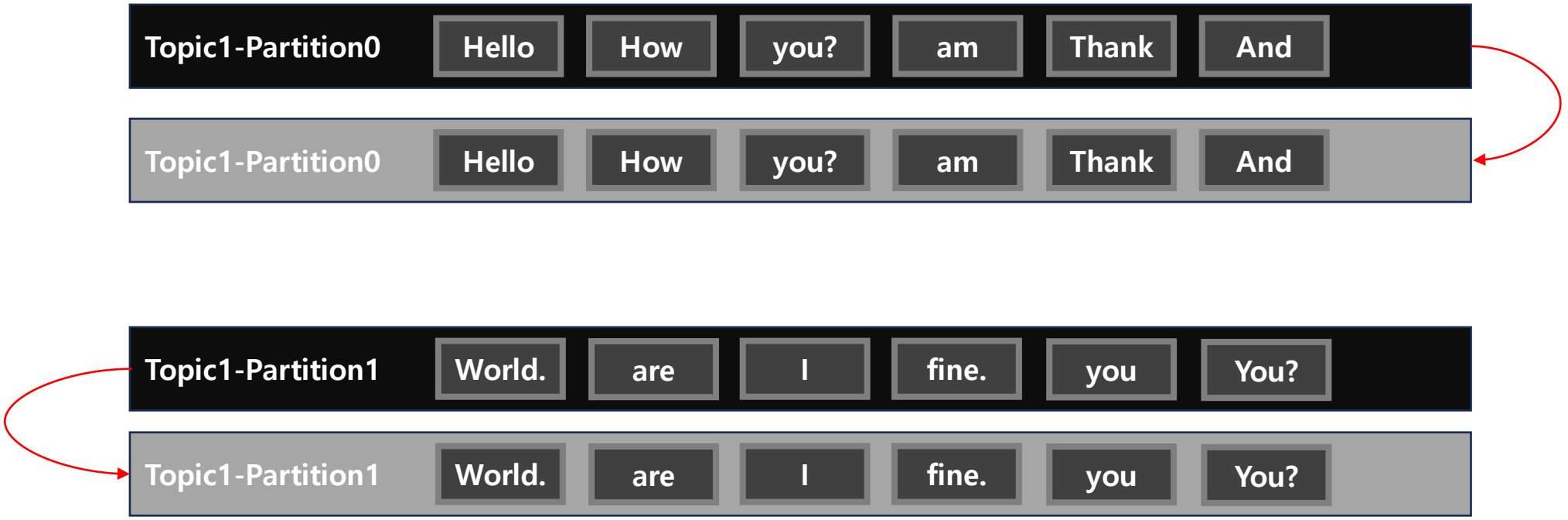
- Kafka의 분산처리 구조 및 장애 회복 동작에 대해 알아봅니다
- 이를 위해서 Partition/Replication에 대해 먼저 알아봅니다
- 2대 이상의 Kafka Broker를 cluster로 묶어서 2개 이상의 Partition/Replication을 구성합니다
- 2개 이상의 Kafka 분산 클러스터로부터 메시지를 수신할 때 유의해야 할 사항에 대해 알아봅니다

Kafka 의 Partition

- 동일한 Topic 내에서의 영역 구분이라고 보시면 됩니다
- Hello world. How are you? I am fine. Thank you. And you? 라는 문장이 아래와 같이 분산되어 저장된 것과 같습니다

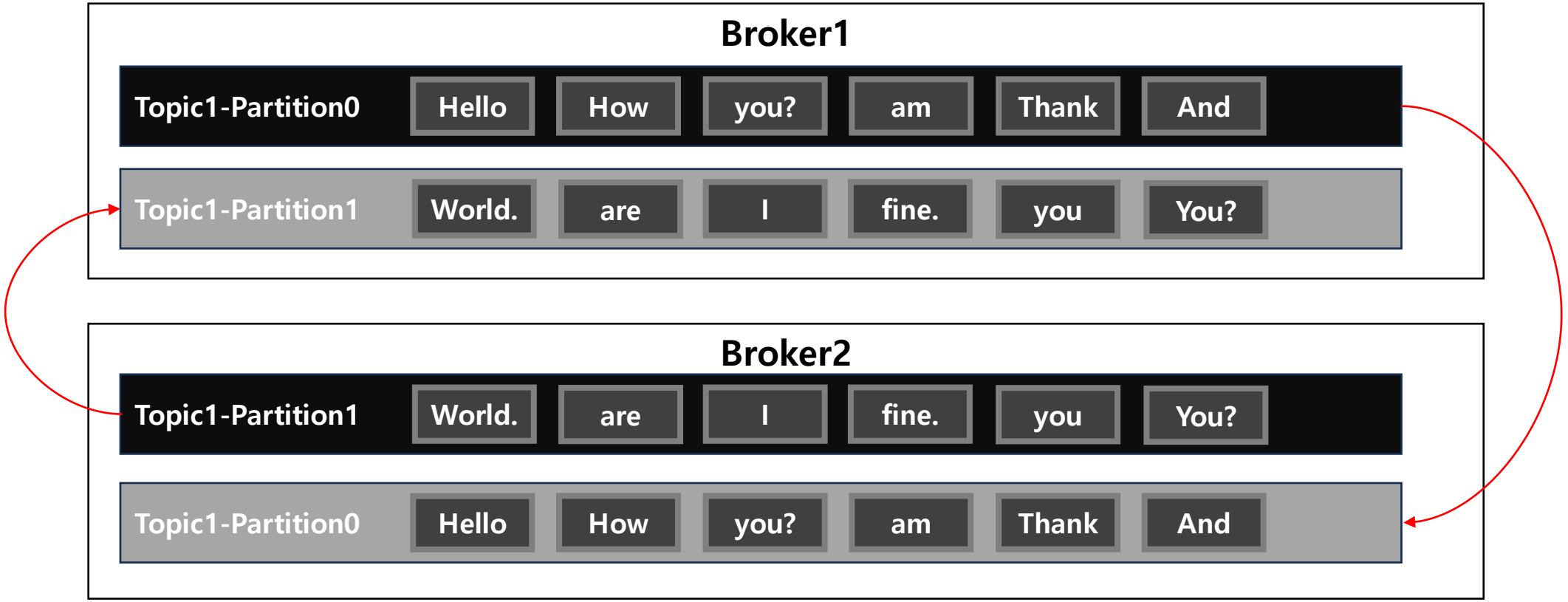
Kafka 의 Replication

- 말 그대로 복제본 입니다
- 복제본을 만드는 이유는 데이터를 저장하고 있는 Broker의 장애를 극복하기 위함 입니다
- 따라서 Kafka 에서 복제본은 동일 Broker가 아닌 다른 Broker에 저장되어야 하겠습니다



Kafka 의 Replication

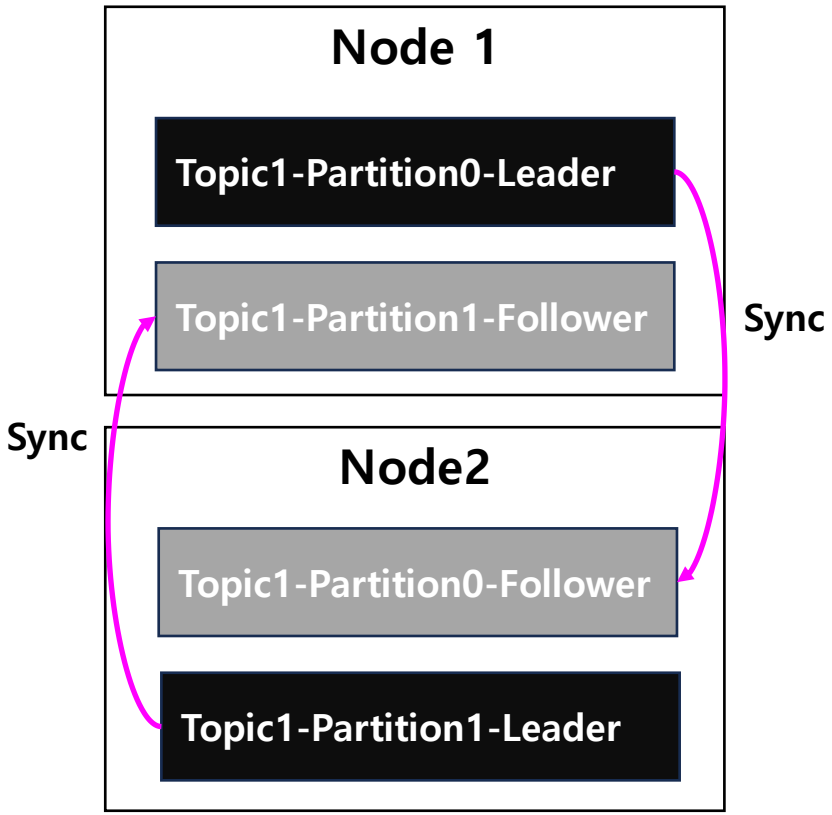
- Kafka 에서 복제본은 동일 Broker가 아닌 다른 Broker에 저장됩니다
- 따라서 아래와 같이 Replication이 배포됩니다



2대의 Kafka broker 구동하기

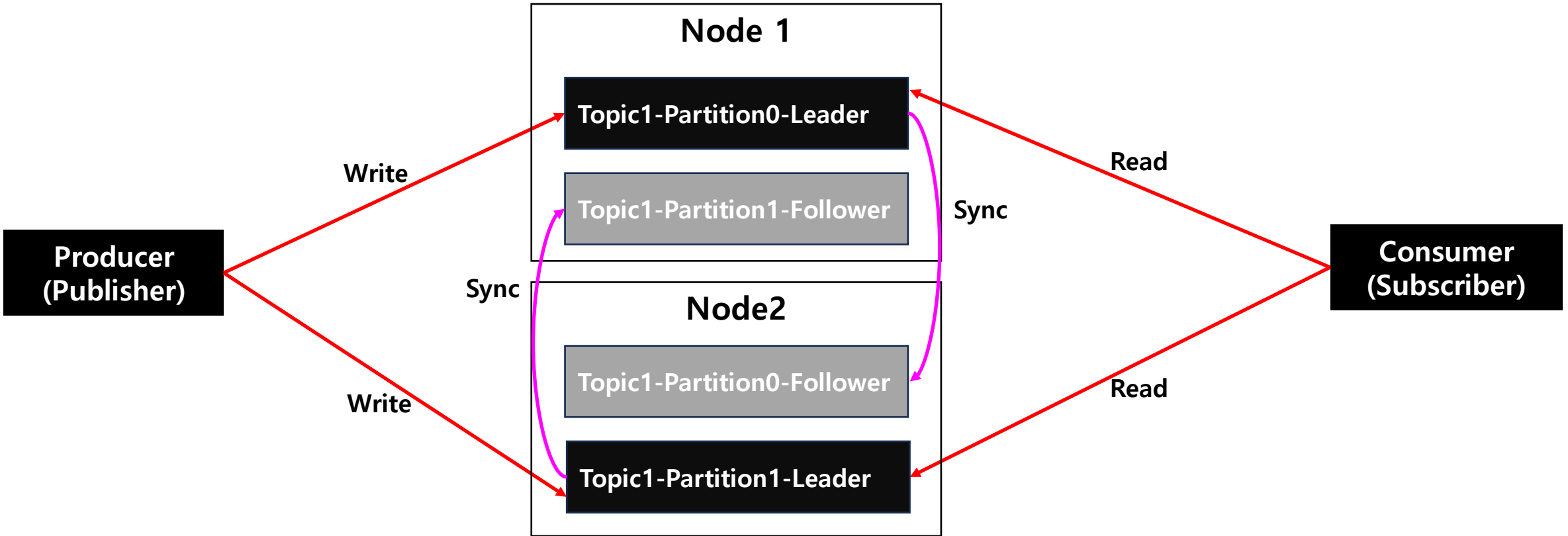
- Cluster 구축을 위해 2대의 Kafka broker를 사용할 것입니다
- Leader Election을 위해 홀수의 노드 수로 구성하는 것이 권장됩니다
- 그러나 단일 PC 내에서 다수의 가상 머신을 구동하는 것은 성능저하를 유발하기 때문에 2대로 진행합니다
- 2대의 Kafka broker를 cluster로 묶으면 오른쪽과 같은 구조가 됩니다

- 동일한 Topic에 대해 2개의 Partition으로 Serving 된다고 가정하면 아래와 같습니다
- Partition0의 Replica가 2개의 Broker에 걸쳐서 배포됩니다
- Partition1의 Replica가 2개의 Broker에 걸쳐서 배포됩니다
- Partition0를 Serving할 Replica(Leader)를 결정 합니다
- Partition1를 Serving할 Replica(Leader)를 결정 합니다
- Partition0의 다른 Replica(Follower)는 Partition0의 Leader Replica로부터 동기화 됩니다
- Partition1의 다른 Replica(Follower)는 Partition1의 Leader Replica로부터 동기화 됩니다



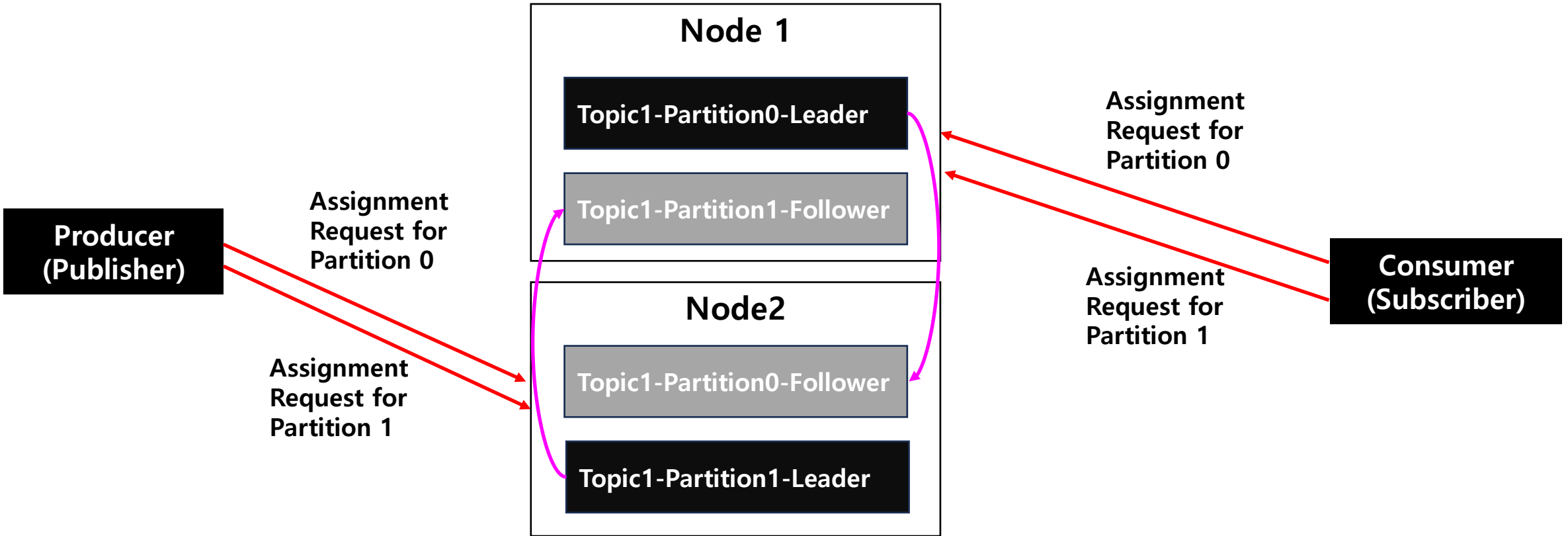
Producer와 Consumer의 Partition 접근

- Producer는 Leader Partition에만 Write 가능합니다
- Consumer는 Leader Partition으로부터만 Read 가능합니다
- Offset management의 일관성을 위해 이러한 구조를 갖습니다



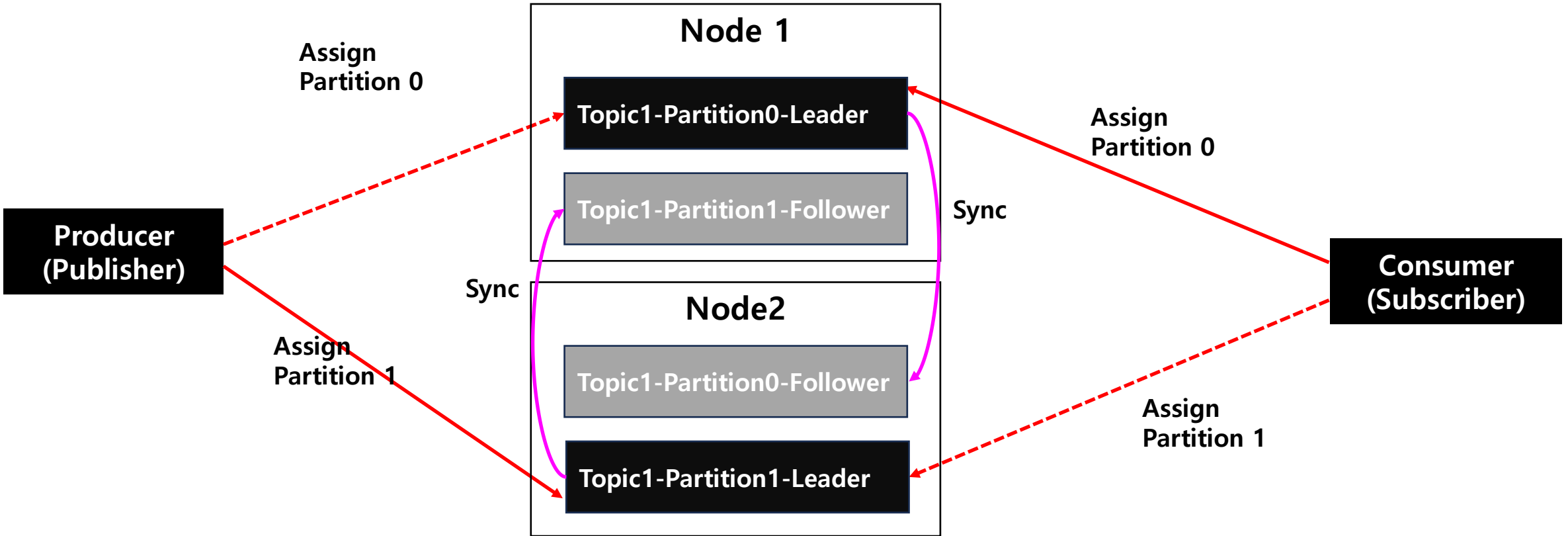
Producer와 Consumer의 Partition 접근

- 이러한 이유로 Read/Write에 앞서서 Partition 할당 요청이 선행됩니다
- Partition 할당 요청은 Broker cluster중 어느 Node에도 가능합니다
- 모든 Kafka Broker는 Partition Leader에 대해 알고 있기 때문입니다



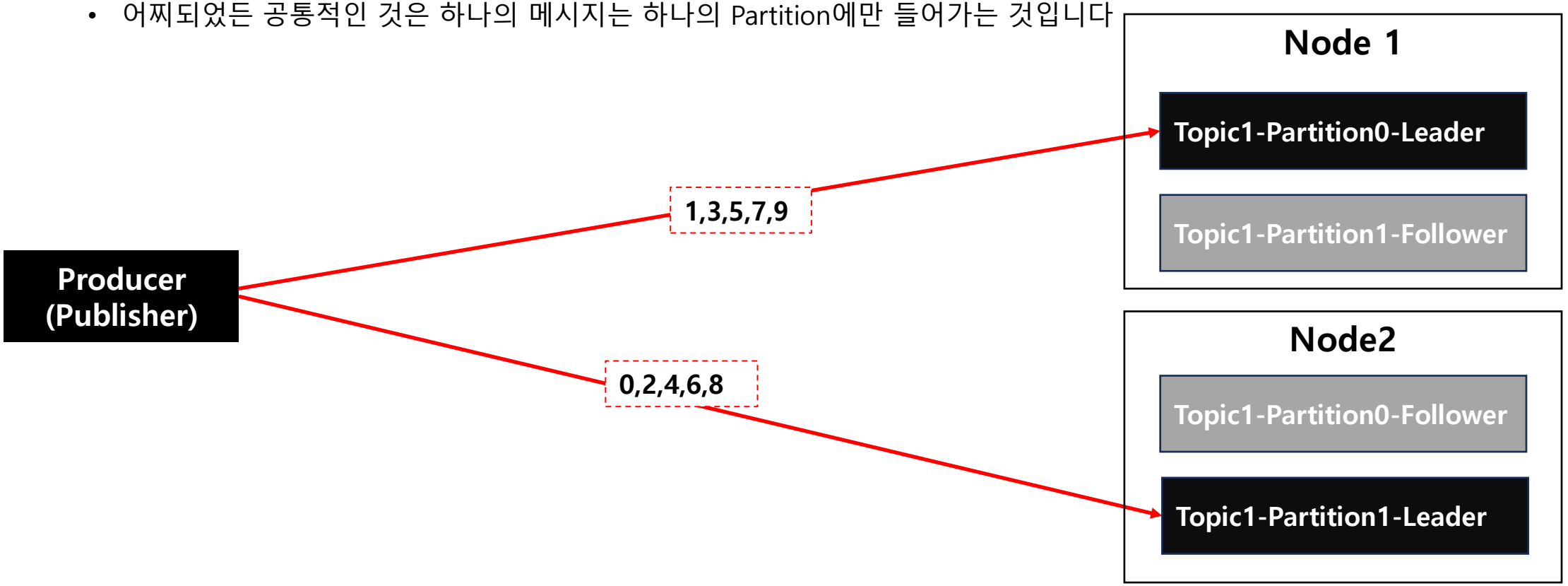
Producer와 Consumer의 Partition 접근

- 따라서 Kafka는 Broker는 각 Partition의 Leader 에게 연결될 수 있도록 정보를 알려줍니다
- 이를 전달받은 Producer/Consumer는 할당된 Broker로 재 연결을 시도합니다



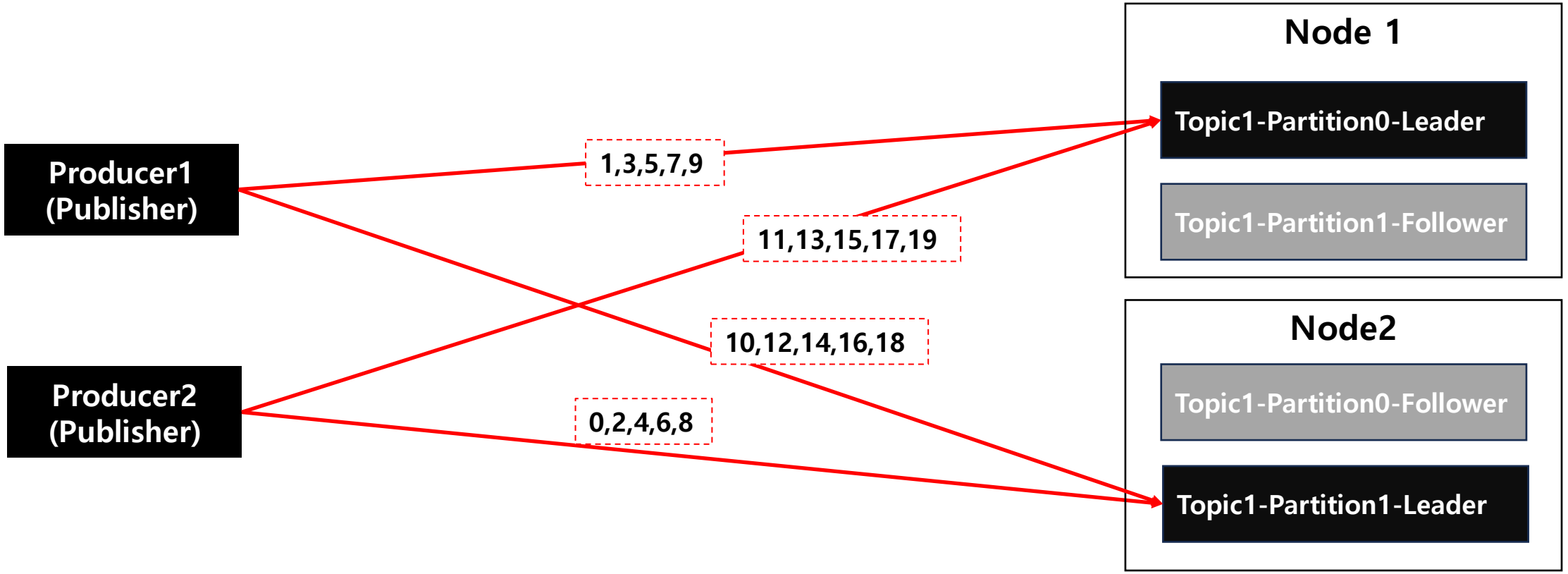
Producer로서의 Partition 할당 정책

- 여러가지 정책이 있지만 어떤 Partition을 선택할지 결정하는 정책일 뿐입니다
- 0 ~ 9 까지의 데이터가 순차적으로 들어간다고 할 때 아래와 같이 round robin 으로 들어갈 수도 있습니다
- 혹은 다른 방법으로 들어갈 수 도 있습니다 (예를 들면 0,1,2,3,4 / 5,6,7,8,9)
- 어찌되었든 공통적인 것은 하나의 메시지는 하나의 Partition에만 들어가는 것입니다



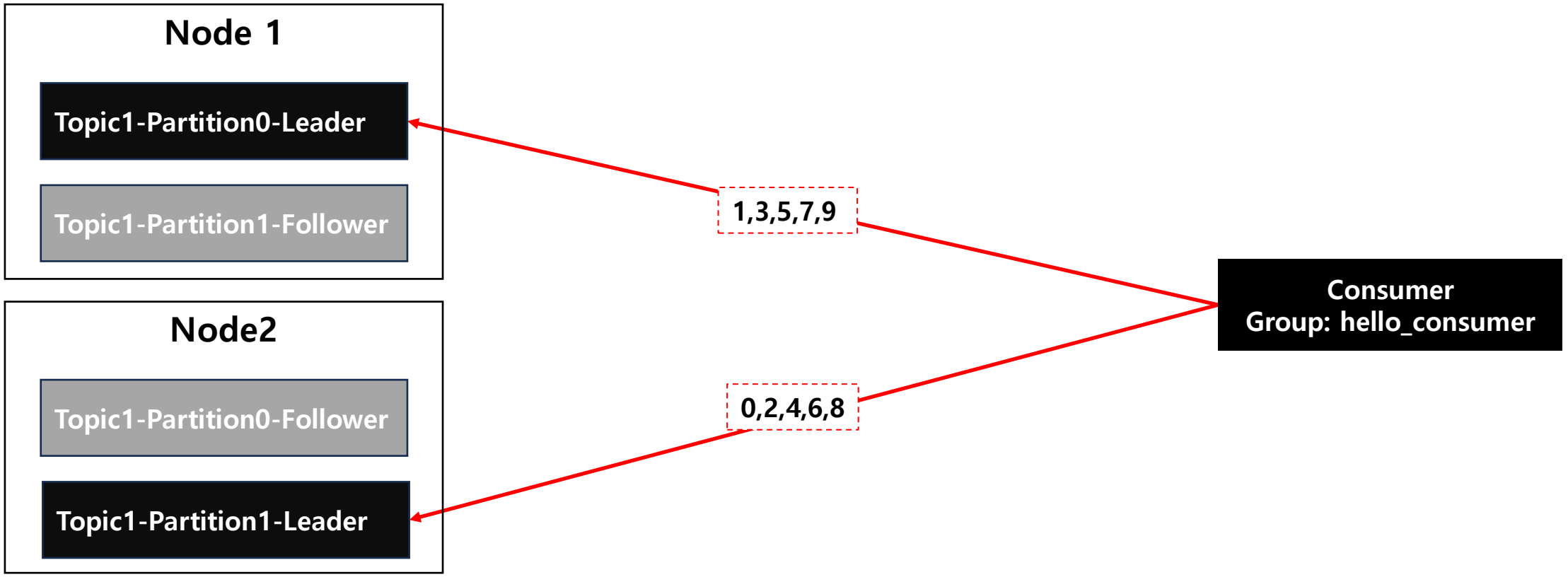
Producer로서의 Partition 할당 정책

- 복수의 Producer의 사용이 가능합니다
- 이 때 각각의 Producer가 어떤 Partition에 할당될지는 정책에 따라 다릅니다
- 그러나 역시 공통적인 것은 하나의 메시지는 하나의 Partition에만 전달 된다는 것입니다



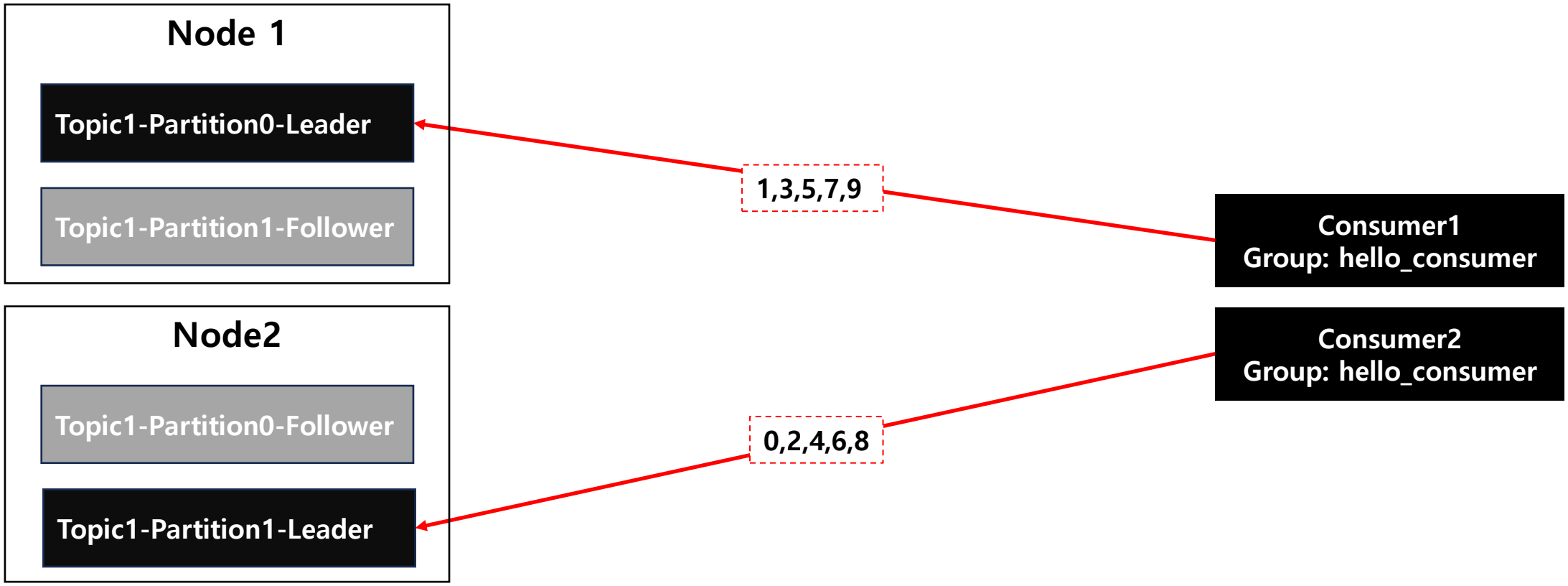
Consumer로서의 Partition 할당 정책

- 하나의 Consumer는 여러 개의 Partition에 할당될 수 있습니다
- Broker에 저장된 메시지는 한번만 처리되고 중복 처리되지 않는다는 관점으로 생각해보면 됩니다
- 단일 Consumer가 여러 개의 Partition을 처리해도 메시지는 중복 처리되지 않습니다



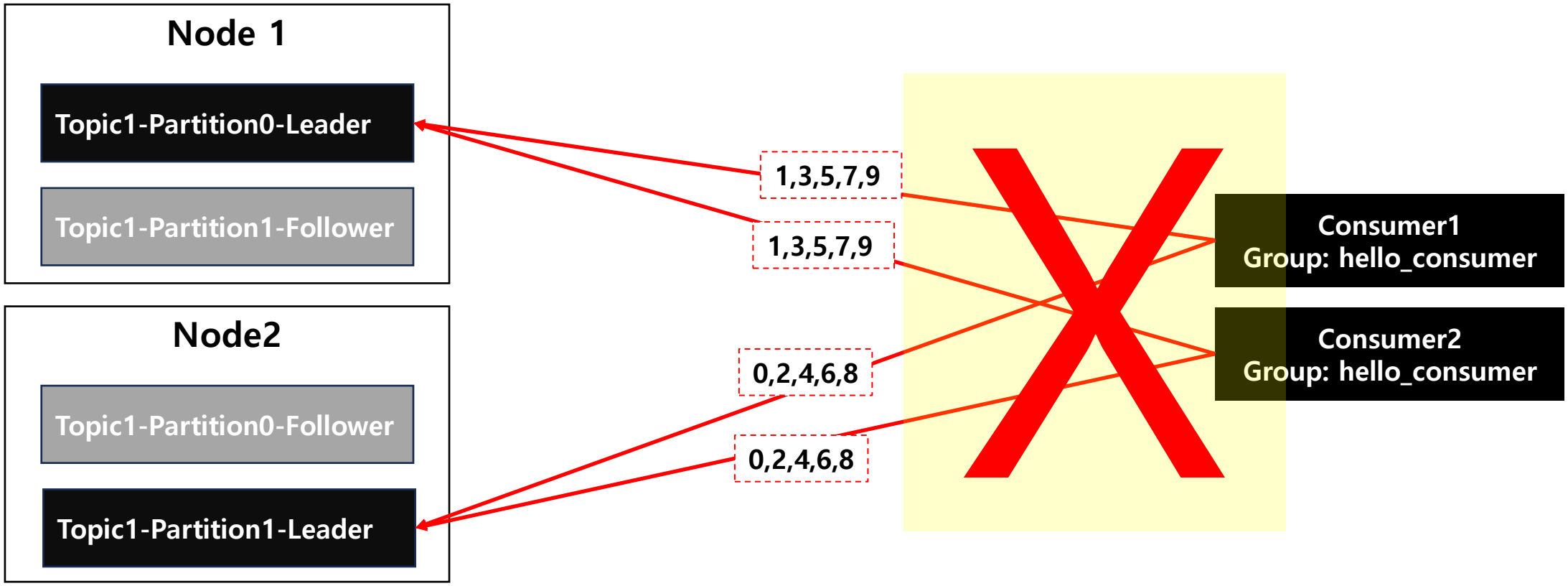
Consumer로서의 Partition 할당 정책

- 복수의 Consumer가 있을 경우 각각은 동일한 Partition에 할당되지 않습니다
- Broker에 저장된 메시지는 한번만 처리되고 중복 처리되지 않는다는 관점으로 생각해보면 됩니다



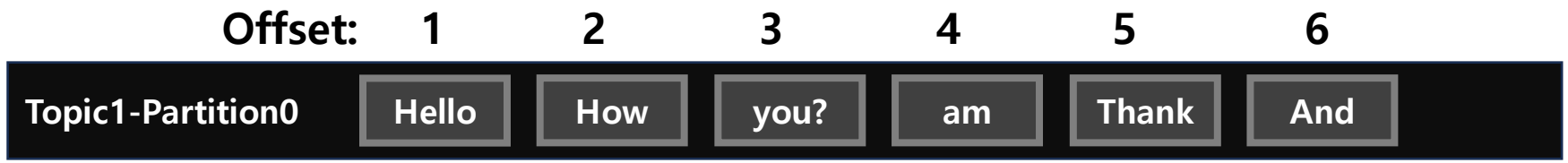
Consumer로서의 Partition 할당 정책

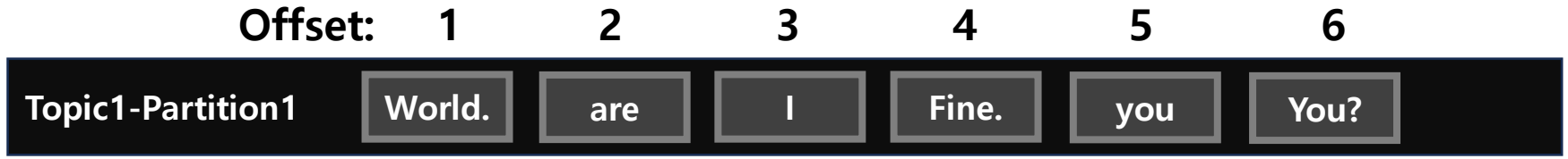
- 복수의 Consumer가 동일한 Partition에 할당되면 메시지는 중복 처리됩니다
- 따라서 Consumer의 Partition 할당은 아래와 같이 이루어지지 않습니다



Kafka 메시지의 offset

- 메시지의 순서를 의미합니다
- Topic에 대해 고유한 것이 아니라 Topic의 Partition에 대해 고유합니다
- 따라서 Kafka 메시지의 offset은 메시지의 순서를 보장하지 않습니다
- 아래의 예제의 Partition0 내에서 Hello → How → you? → am → Thank → And 의 순서는 보장됩니다
- 아래의 예제의 Partition1 내에서 World. → are → I → fine → you → You? 의 순서는 보장됩니다
- 그러나 Hello world. How are you? I am fine. Thank you. And you? 의 순서는 보장되지 않습니다





Kafka Consumer의 offset management

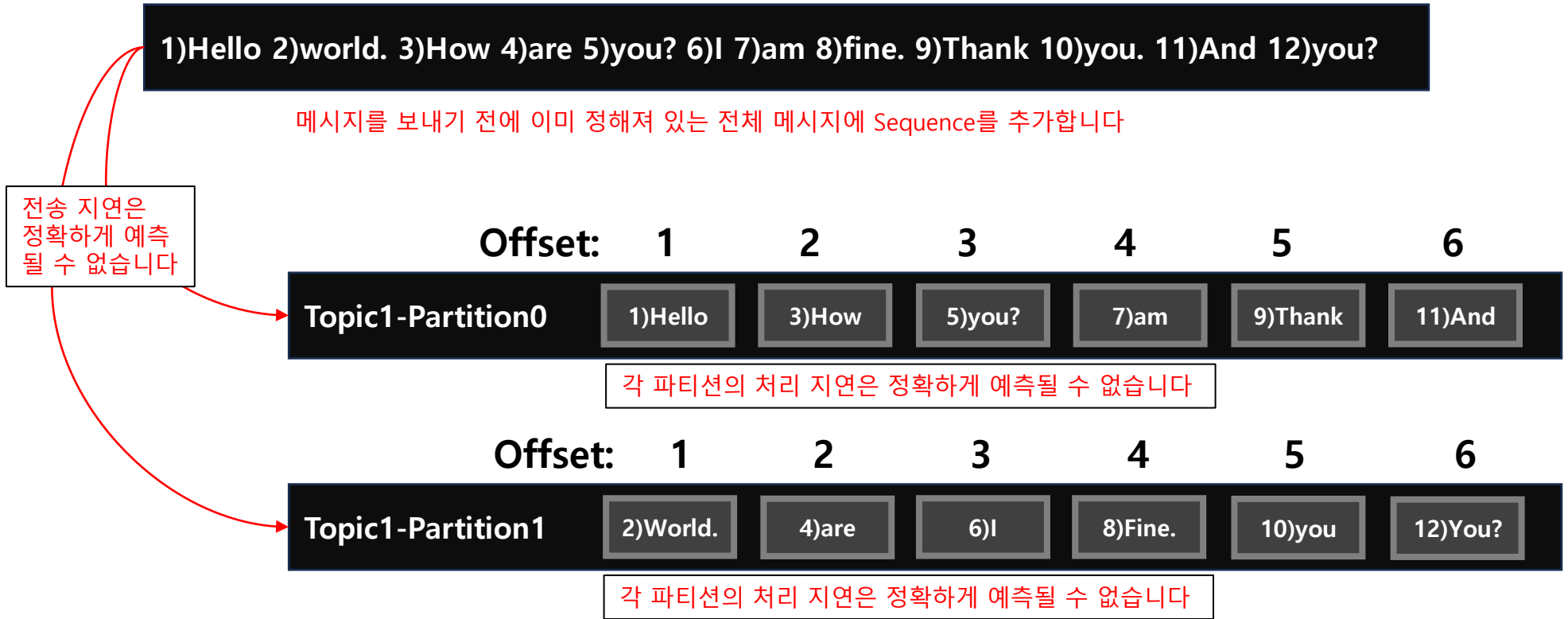
- Kafka Consumer가 Topic 메시지의 offset을 어디까지 수신했는지는 Partition 기준으로 관리됩니다
- ***모든 Partition을 합친 전체 메시지 중에서 몇 번째 offset 까지 수신했는지는 보장되지 않으며 알 수 없습니다***
- 이것을 이해하기 위해서 다음 슬라이드를 확인하겠습니다

Kafka Consumer의 offset management

- 전체 메시지의 순서는 각 Partition에 도달한 순서가 아니며 메시지를 보내기 전에 이미 결정되어 있다는 점을 생각해봅니다
- 메시지를 Topic-Partition에 보낸 후에는 전송지연, 처리지연이 발생합니다
- 따라서 Partition에 최종 저장된 시점에서의 시간 순서대로 전체 메시지를 나열해도 최초의 메시지 순서를 보장할 수 없습니다

Kafka Consumer의 offset management

- 그러면 우리가 해볼 수 있는 것은 이미 메시지가 정해진 그 시점에서 sequence를 달아서 보내는 것입니다
- 인위적으로 sequence를 달아서 보낼 수도 있습니다
- 그런데 이미 비즈니스 로직 상에서 데이터의 발생시점을 알 수 있다면 그 필드를 붙여서 보내도 됩니다(created, updated 필드)



Kafka Consumer의 offset management

- 전송지연, 처리지연은 Consumer에서도 발생합니다
- 따라서 Consumer가 처리한 메시지의 순서는 Producer가 보낸 메시지의 순서와 동일하지 않습니다
- 그러나 ***메시지에 별도로 붙여서 보낸 Sequence에 의해 re-ordering***이 가능합니다

중간 정리

- ***Partition***
- Kafka의 메시지를 Partition에 분산해서 저장할 수 있습니다
- 어떤 정책이든 하나의 메시지는 하나의 Partition에 들어가는 것은 동일합니다

- ***Replication***
- Kafka의 메시지의 Partition은 복제되어 다른 Broker에 저장할 수 있습니다
- 몇 개를 복제하든 Partition의 Leader에만 Read/Write operation이 가능합니다

- ***Message offset***
- 메시지를 보내고 받으면서 전송지연, 처리지연이 발생합니다
- 전송지연, 처리지연이 발생해도 Partition 내에서는 메시지가 도달한 순서대로 그 순서가 보장됩니다
- 그러나 분산 저장된 Partition으로부터 메시지를 수신하여 취합하면 원래 메시지의 순서는 보장되지 않습니다

Partition Leader Election

- 1개 또는 그 이상의 Replication 묶음 중에서 누구를 Leader로 정할지에 대한 결정 입니다
- Election 이므로 과반의 찬성이 발생하는 구조여야 합니다
- ***Voter로서의 역할을 하는 노드의 수는 홀수로 구성되어야 합니다***
- 1대, 3대, 5대....

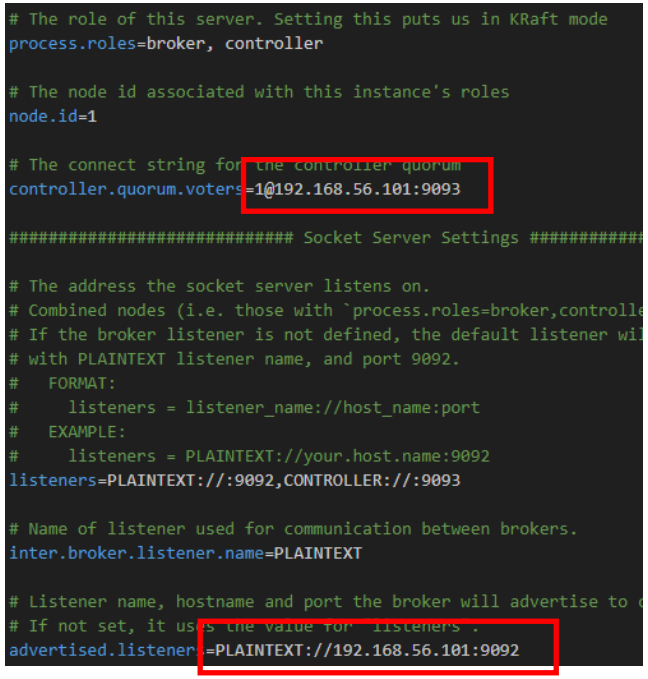
- 이것은 Broker의 role에서 하는 것이 아니라 별도의 role 에서 진행합니다
- 우리는 그것을 KRaft 라는 것을 이용합니다
- Broker설정에서는 이를 controller(voter) 라고 합니다

- 결국 Broker의 role은 아래의 2개 입니다
- Broker = broker
- KRaft = controller(voter)

- 이제 https://kafka.apache.org/documentation/#kraft 문서를 확인합니다

KRaft 설정(기존의 Kafka Broker)

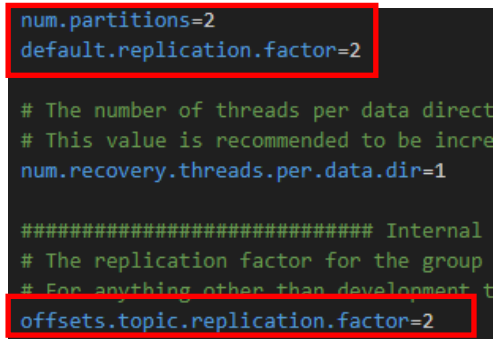
- 설정 파일의 경로는 Kafka home 디렉토리를 기준으로 config/kraft/server.properties 에 있습니다
- 기존에 우리가 Kafka broker 구동시 사용했던 설정파일 입니다
- controller role이 부여된 Broker는 9093 port를 추가로 수신합니다
- controller role이 부여된 Broker는 Partition Leader Election의 voter로 활동할 수 있습니다
- Voter의 id는 node.id 입니다



1@192.168.57.101:9093의 의미

1) voter(controller)의 id는 node.id와 동일한 1 입니다
2) process.roles에 controller가 추가되어 있습니다
3) listeners 설정에 의해 controller는 9093 port를 수신합니다
4) 그러므로 node 에서는 controller로서 9093 port를 수신하고 있을 것입니다

따라서 voter(controller) 의 주소는 1@<현재 node의 ip>:9093 입니다



추가로 partition/replication 설정을 합니다

Consumer offset management용 Topic이 별도로 있습니다
이에 대한 replication 설정을 합니다

KRaft 설정(새로운 Kafka Broker)

- 홀수의 voter 수를 유지하기 위해 새로운 Kafka Broker에서는 controller(voter) 로서의 역할은 수행하지 않습니다
- 새로운 Kafka Broker이므로 node.id는 고유한 다른것으로 바꿉니다
- 이 2대의 Kafka 클러스터에서 유일한 voter(controller)의 주소는 기존의 Kafka Broker 뿐입니다

```
# The role of this server. Setting this puts us in KRa
process.roles=broker

# The node id associated with this instance's roles
node.id=2

# The connect string for the controller quorum
controller.quorum.voters=1@192.168.56.101:9093

############################ Socket Server Settings #

# The address the socket server listens on.
# Combined nodes (i.e. those with `process.roles=broke
# If the broker listener is not defined, the default l
# with PLAINTEXT listener name, and port 9092.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092

# Name of listener used for communication between brok
inter.broker.listener.name=PLAINTEXT

# Listener name, hostname and port the broker will adv
# If not set, it uses the value for "listeners"
advertised.listeners=PLAINTEXT://192.168.57.101:9092
```

Advertised의 의미: Partition assignment request 에서 알려주는 Broker 주소
따라서 외부에서 접근 가능한 IP로 advertised 되어야 합니다

```
num.partitions=2
default.replication.factor=2

# The number of threads per data direct
# This value is recommended to be incre
num.recovery.threads.per.data.dir=1

############################ Internal
# The replication factor for the group
# For anything other than development t
offsets.topic.replication.factor=2
```

추가로 partition/replication 설정을 합니다

Consumer offset management용 Topic이 별도로 있습니다
이에 대한 replication 설정을 합니다

Kafka 구동

- 기존 Kafka Broker의 구동
- 2가지 선택지가 있습니다
- 1) Kafka cluster ID를 확인만 하고 storage 포맷은 하지 않음
- 2) Kafka cluster ID를 새로 만들고 storage를 포맷

```
cat /tmp/kraft-combined-logs/meta.properties  | grep cluster.id
./bin/kafka-server-start.sh -daemon config/kraft/server.properties
```

```
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$ cat /tmp/kraft-combined-logs/meta.properties  | grep cluster.id
cluster.id=ILA_jE0YTqCofpAro5ksiQ
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$
```

- 새로운 Kafka Broker에서는 기존 kafka storage를 삭제하고 포맷을 반드시 진행해줍니다
- 이 때 cluster id를 기존 Kafka와 동일하게 해야 동일한 cluster로 묶입니다

```
rm –rf /tmp/kraft-combined-logs
KAFKA_CLUSTER_ID=ILA_jE0YTqCofpAro5ksiQ
./bin/kafka-storage.sh format -t $KAFKA_CLUSTER_ID -c config/kraft/server.properties
./bin/kafka-server-start.sh -daemon config/kraft/server.properties
```

Kafka 구동

- 기존의 Broker에서는 아래와 같이 9092, 9093 port를 수신하고 있는지 확인합니다
- 또한 새로운 Broker로부터 voter 연결이 되었는지 확인합니다

```
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$ jps
19892 Kafka
20006 Jps
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$ netstat -nap | grep 19892
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp6       0      0 :::42053                :::*                    LISTEN      19892/java
tcp6       0      0 :::9092                 :::*                    LISTEN      19892/java
tcp6       0      0 :::9093                 :::*                    LISTEN      19892/java
tcp6       0      0 192.168.56.101:33306    192.168.56.101:9093     ESTABLISHED 19892/java
tcp6       0      0 192.168.56.101:9093     192.168.56.1:61901      ESTABLISHED 19892/java
tcp6       0      0 192.168.56.101:9093     192.168.56.101:33312    ESTABLISHED 19892/java
tcp6       0      0 192.168.56.101:33312    192.168.56.101:9093     ESTABLISHED 19892/java
tcp6       0      0 192.168.56.101:9093     192.168.56.101:33306    ESTABLISHED 19892/java
tcp6       0      0 192.168.56.101:9093     192.168.56.1:61902      ESTABLISHED 19892/java
unix  2      [ ]         STREAM     CONNECTED     70412    19892/java
unix  2      [ ]         STREAM     CONNECTED     70410    19892/java
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$
```

- 새로운 Broker에서는 아래와 같이 9092 port만을 수신하고 있는지 확인합니다
- 또한 기존의 Broker에 있는 voter로 연결이 되었는지 확인합니다

```
ubuntu@ubuntu-server:~$ jps
12110 Jps
11567 Kafka
ubuntu@ubuntu-server:~$ netstat -nap | grep 11567
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp6       0      0 :::9092                 :::*                    LISTEN      11567/java
tcp6       0      0 :::39881                :::*                    LISTEN      11567/java
tcp6       0      0 10.0.2.15:52216         192.168.56.101:9093     ESTABLISHED 11567/java
tcp6       0      0 10.0.2.15:52390         192.168.56.101:9093     ESTABLISHED 11567/java
unix  2      [ ]         STREAM     CONNECTED     66479    11567/java
unix  2      [ ]         STREAM     CONNECTED     66477    11567/java
ubuntu@ubuntu-server:~$
```

여기까지 되었으면
Kafka 분산 클러스터 완성 입니다

분산클러스터에 Pub/Sub 테스트

- 기존에 사용했던 Producer/Consumer 코드를 재사용 합니다
- 단, bootstrap_servers 설정이 변경됩니다
- 실제 가상머신 2대에 할당된 호스트 전용 어댑터 IP(Virtualbox) 또는 Shared 네트워크 어댑터 IP(UTM) 를 기록합니다

```python
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='192.168.56.101:9092,192.168.57.101:9092')

try:
    while True:
        message = input("Enter message: ")
        producer.send('helloworld', message.encode('utf-8'))
finally:
    producer.close()
```

Producer를 구동하고
메시지를 입력해봅니다

```python
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    'helloworld',
    bootstrap_servers='192.168.56.101:9092,192.168.57.101:9092',
    group_id='mygroup',
    auto_offset_reset='earliest'
)

for message in consumer:
    print(f"Received: {message.value.decode('utf-8')}")

consumer.close()
```

Consumer를 구동하고 수신되는
메시지를 확인해봅니다

Kafka topic 상태 확인

- 아래와 같이 topic의 상태에 대해 확인해봅니다
- Leader:1, Leader:2 는 Broker의 ID를 의미합니다
- Partition0는 Broker1이 Leader, Partition1는 Broker2가 Leader임을 의미합니다
- Replicas는 Broker1과 Broker2에 걸쳐서 분포되어 있습니다
- Replicas의 순서가 다른것은 Leader가 각각 다르기 때문입니다
- Isr(In Sync Replicas): 동기화가 되어 있는 Replicas 입니다

```
./bin/kafka-topics.sh --describe --topic helloworld --bootstrap-server 192.168.56.101:9092,192.168.57.101:9092
```

```
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$ ./bin/kafka-topics.sh --describe --topic helloworld --bootstrap-server 192.168.56.101:9092,192.168.57.101:9092
Topic: helloworld       TopicId: VfQhnV5CRgy4sIKeJTXfzg PartitionCount: 2       ReplicationFactor: 2    Configs: segment.bytes=1073741824
        Topic: helloworld       Partition: 0    Leader: 1       Replicas: 1,2   Isr: 1,2        Elr:    LastKnownElr:
        Topic: helloworld       Partition: 1    Leader: 2       Replicas: 2,1   Isr: 2,1        Elr:    LastKnownElr:
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$
```

Kafka consumer 상태 확인

- Partition 1에 총 2개의 메시지가 있습니다
- Partition 0에 총 5개의 메시지가 있습니다
- Consumer 1개가 2개의 Partition의 메시지를 수신하고 있습니다
- Consumer는 2개의 Partition에 있는 latest 메시지까지 모두 수신하였습니다

```
./bin/kafka-consumer-groups.sh --bootstrap-server 192.168.56.101:9092,192.168.57.101:9092 --group mygroup --describe
```

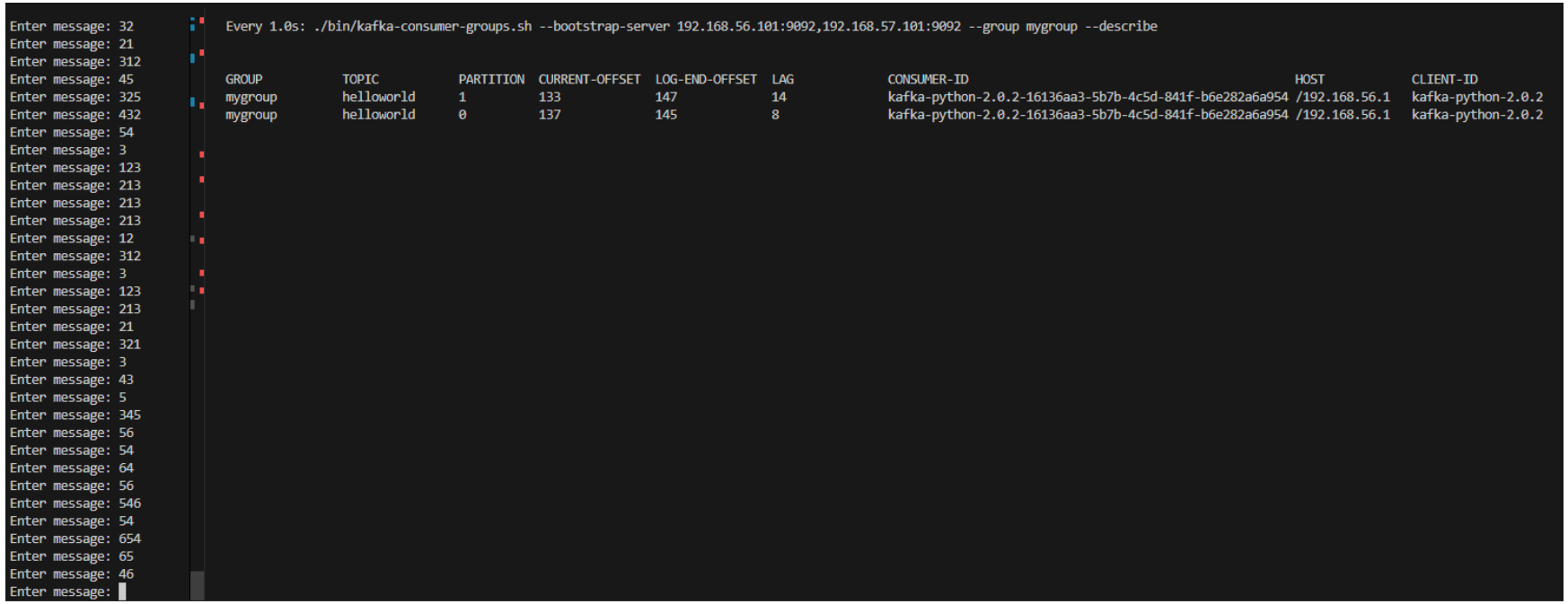
```
ubuntu@ubuntu-server:~/kafka_2.13-3.8.0$ ./bin/kafka-consumer-groups.sh --bootstrap-server 192.168.56.101:9092,192.168.57.101:9092 --group mygroup --describe

GROUP           TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID                                     HOST            CLIENT-ID
mygroup         helloworld      1          2               2               0               kafka-python-2.0.2-16136aa3-5b7b-4c5d-841f-b6e282a6a954 /192.168.56.1   kafka-python-2.0.2
mygroup         helloworld      0          5               5               0               kafka-python-2.0.2-16136aa3-5b7b-4c5d-841f-b6e282a6a954 /192.168.56.1   kafka-python-2.0.2
```

Consumer offset 모니터링

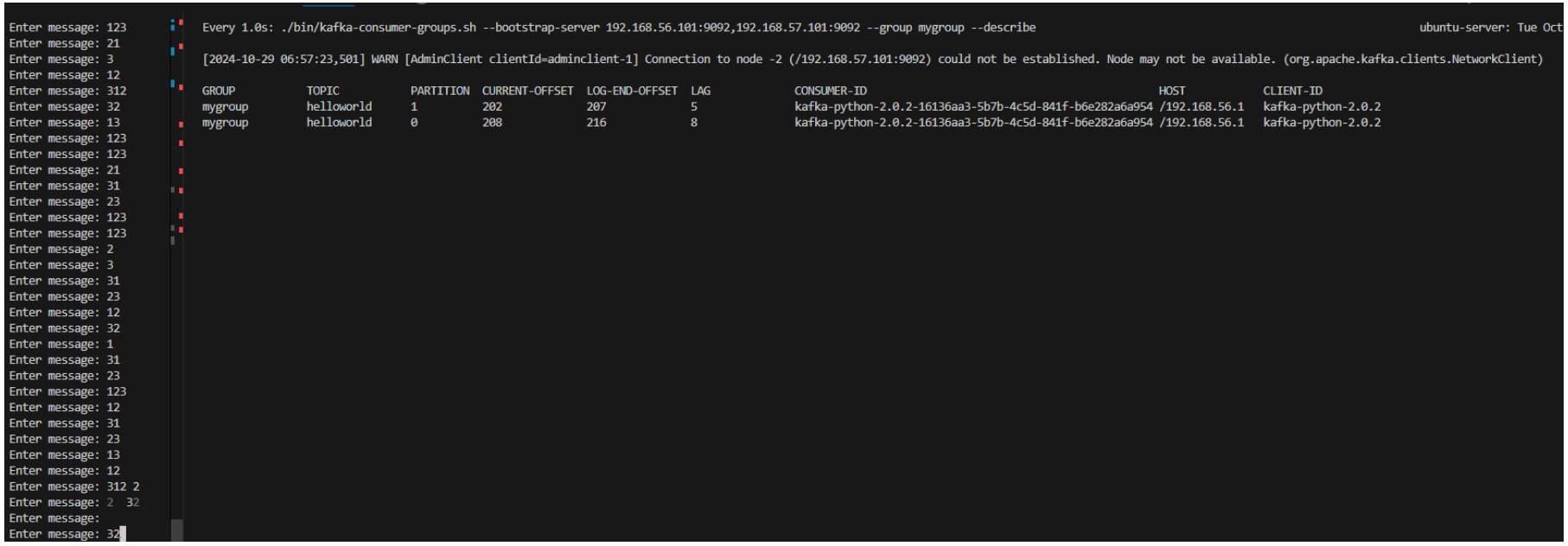
- watch커맨드를 이용해 1초 간격으로(-n 1) Consumer offset 모니터링 커맨드를 실행합니다
- Producer 터미널과 아래 커맨드를 입력한 터미널을 join 시킵니다
- Producer에 메시지를 계속 입력하면서 Partition별로 latest offset과 consumer offset을 확인합니다

```
watch -n 1 "./bin/kafka-consumer-groups.sh --bootstrap-server 192.168.56.101:9092,192.168.57.101:9092 --group mygroup --describe"
```

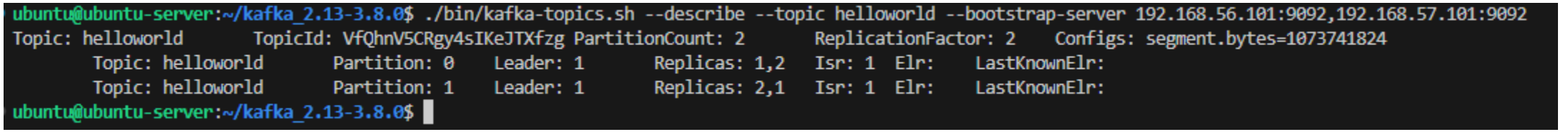


Broker 2의 셧다운

- Broker 2의 Kafka 프로세스를 kill -9 커맨드로 강제 종료시킵니다
- 비록 Broker 2에는 연결 실패했으나 메시지의 Pub/Sub은 정상적으로 수행되고 있는 것을 확인할 수 있습니다



- Broker 2가 Leader였던 Partition1을 Broker1이 이어받았습니다



정리

- 2대의 Broker role과 1대의 controller role로 Kafka cluster를 구성하여 테스트 해보았습니다
- 2대의 Partition으로 메시지가 분산 저장되는 것을 확인하였습니다
- 이 중에서 1대 Broker의 셧다운이 Replica에 의해 회복되는 것을 확인하였습니다

- 이 구성은 권장구성은 아닙니다
- 1대의 controller role을 가진 노드가 단일실패점(SPOF) 가 되기 때문입니다
- 그럼에도 불구하고 controller role을 가진 노드는 voter로서 홀수로 구성해야 하기 때문에 3개가 권장되는 것입니다