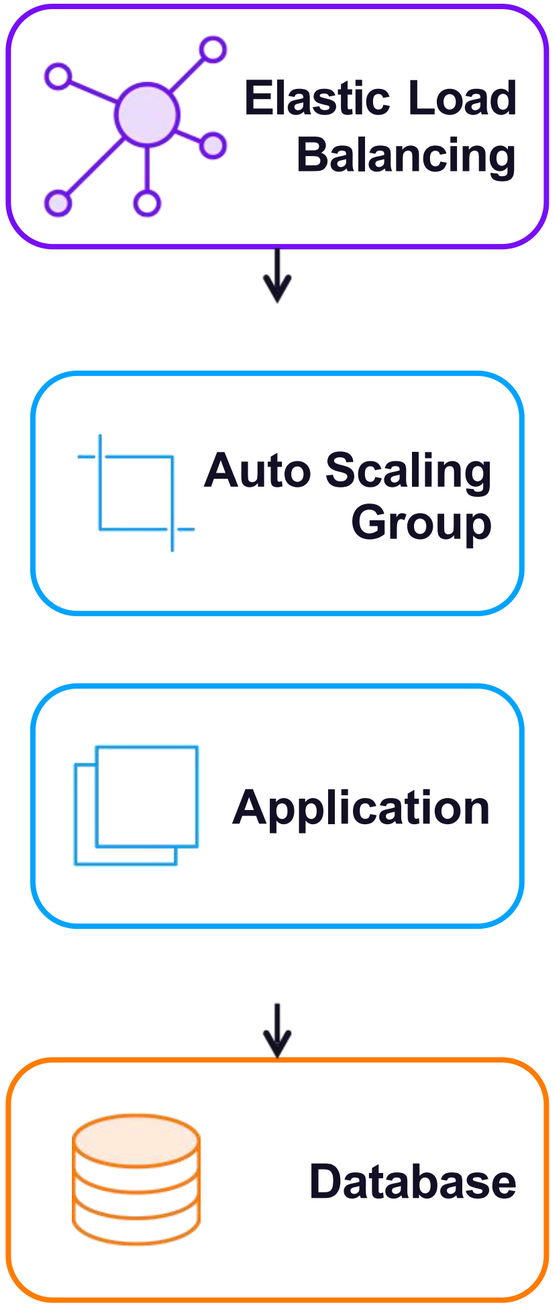
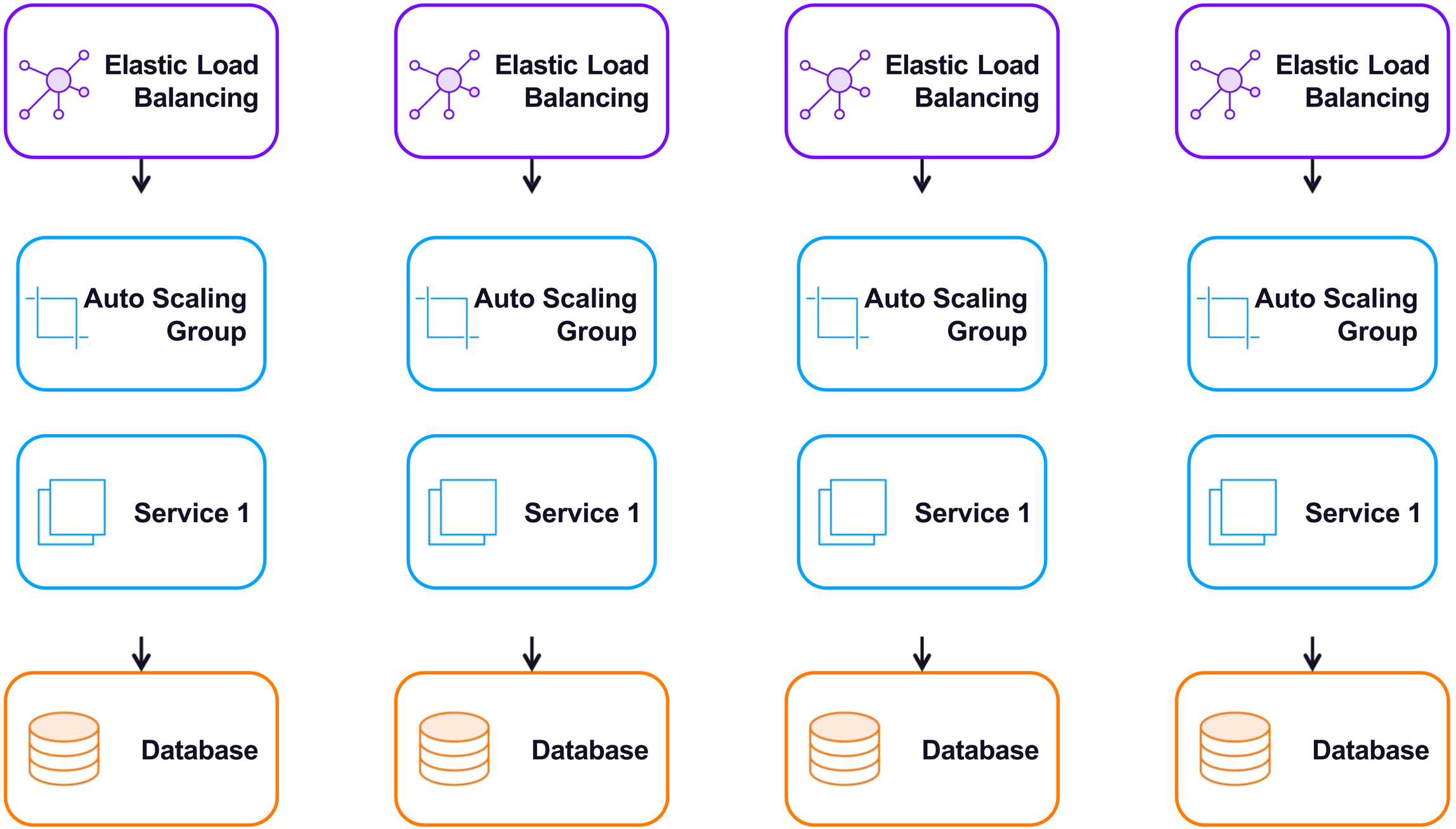
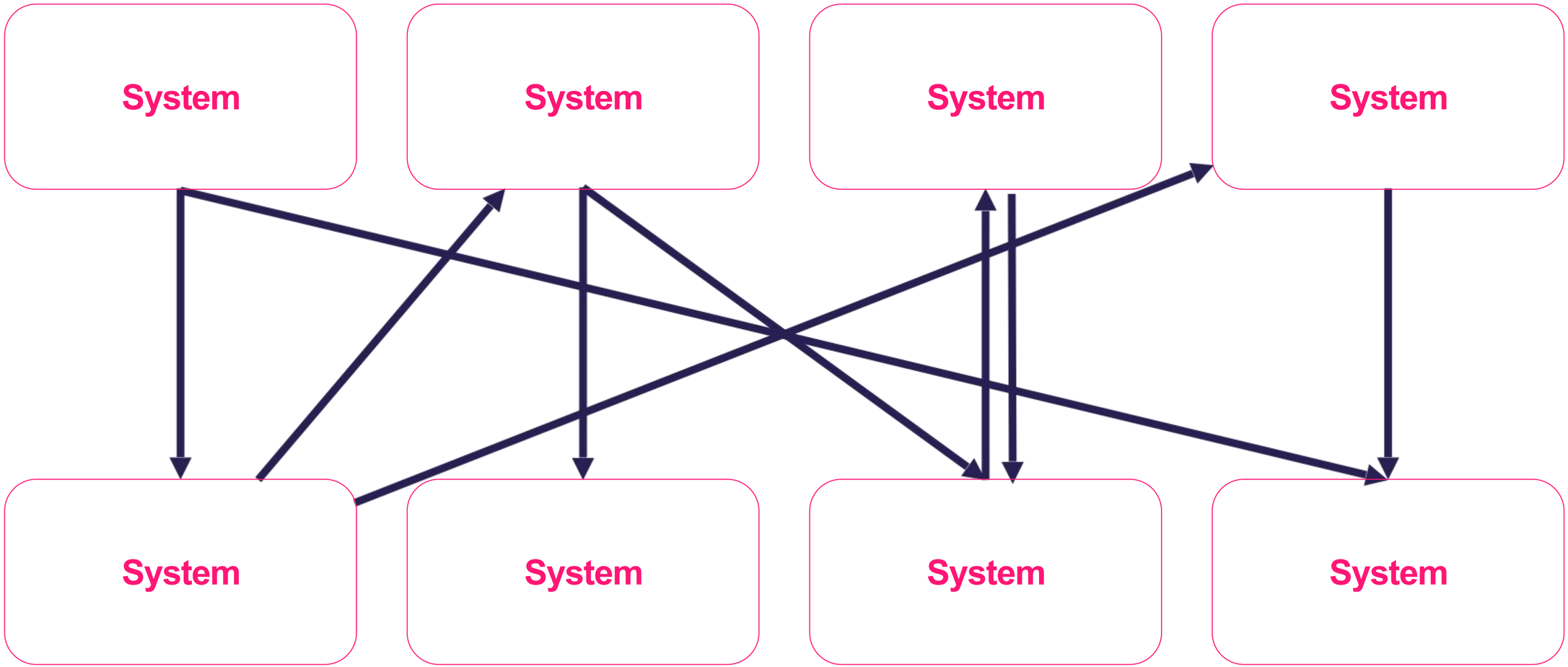
Introducing Pub Sub Systems

Meeting Pub-Sub Systems


Elastic Load Balancing
Auto Scaling Group
Application
Database


Elastic Load Balancing
Elastic Load Balancing
Elastic Load Balancing
Elastic Load Balancing
Auto Scaling Group
Auto Scaling Group
Auto Scaling Group
Auto Scaling Group
Service 1
Service 1
Service 1
Service 1
Database
Database
Database
Database

Entangled System




Kafka


Items Service


UI Service

We Call:

The entity/app creating a message, a publisher or producer

The entity/app consuming messages from a channel as a consumer

The system where the channels live and handle this requests as an Event Bus or, more recently, streaming platforms

The channel where messages flow as channel or topic

Back to Definitions

We say the pub-sub system is reliable when you ensure there is no message loss

Has at most once processing when you ensure there is no message duplication

And has exactly-once processing when you only process a message once ensuring it wasn’t lost. Of course, the holy grail.

We Had Ton of Other Pub/Sub Systems Before:

RabbitMQ

Mulesoft

Redis

Amazon SQS

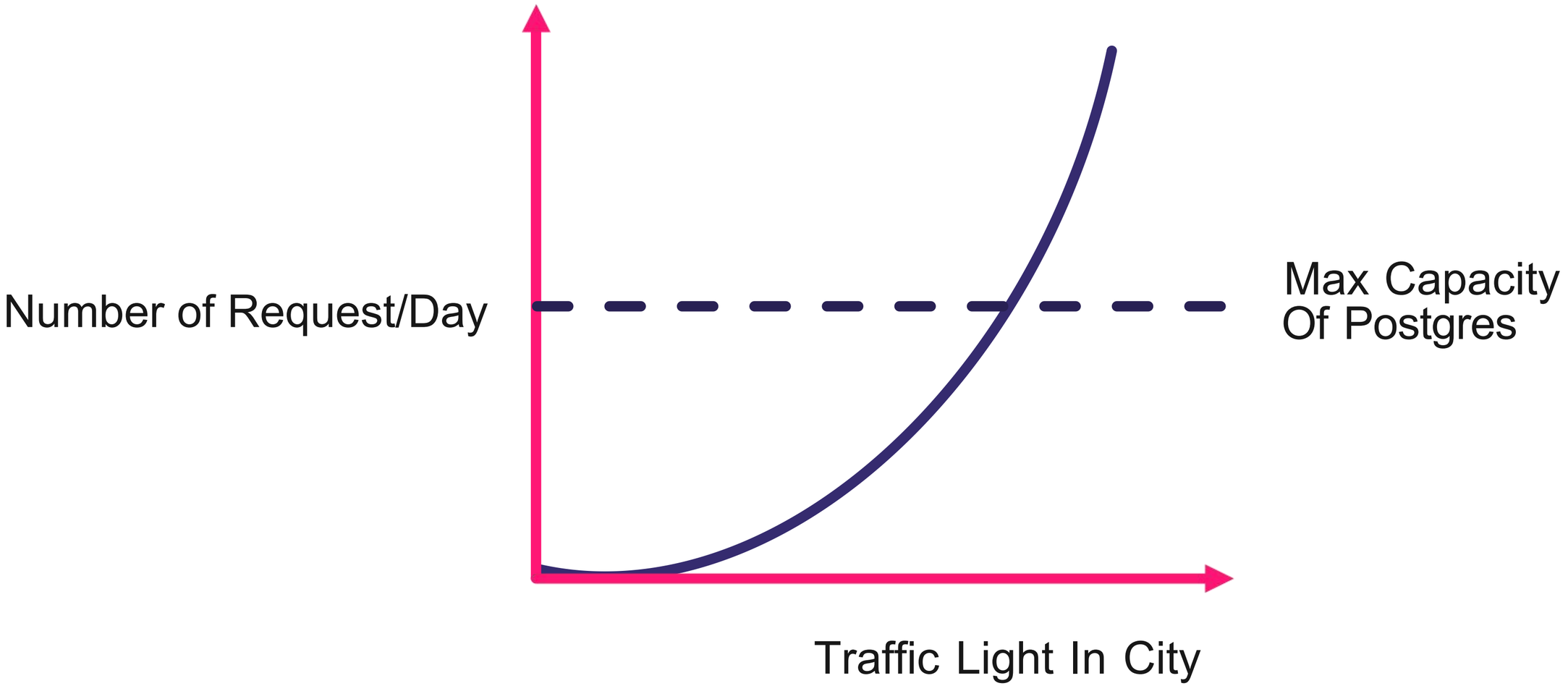
Amazon SNS

Azure HDInsight, Azure Streaming Service, and Google Pub/Sub

Why Kafka?

A Case Study: Traffic Lights

A Case Study: Traffic Lights

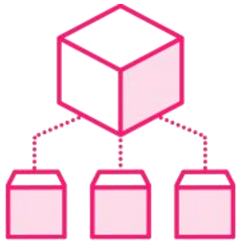


A Case Study: Traffic Lights

A Case Study: 911 Systems

Kafka comes to solve this!

What Is Kafka?



Event Streaming Plataform

Scalable

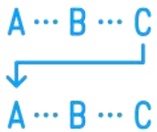
High Throughput

Difference between Kafka and the Rest of Solutions

RabbitMQ: As Kafka uses topics in an unidirectional log without handshake, the difference in performance is brutal, easily 1000x

Mulesoft: Kafka makes it simple to create a channel of communication simply by producing a message the topic will be autocreated

Redis: Redis does not handle streaming data nor has a history past messages



Amazon SQS: It based on queues, way better tan RabbitMQ but still there is an order of magnitude in throughput and speed between Kafka and SQS. Plus SQS is not exactly cheap

Kafka Architecture

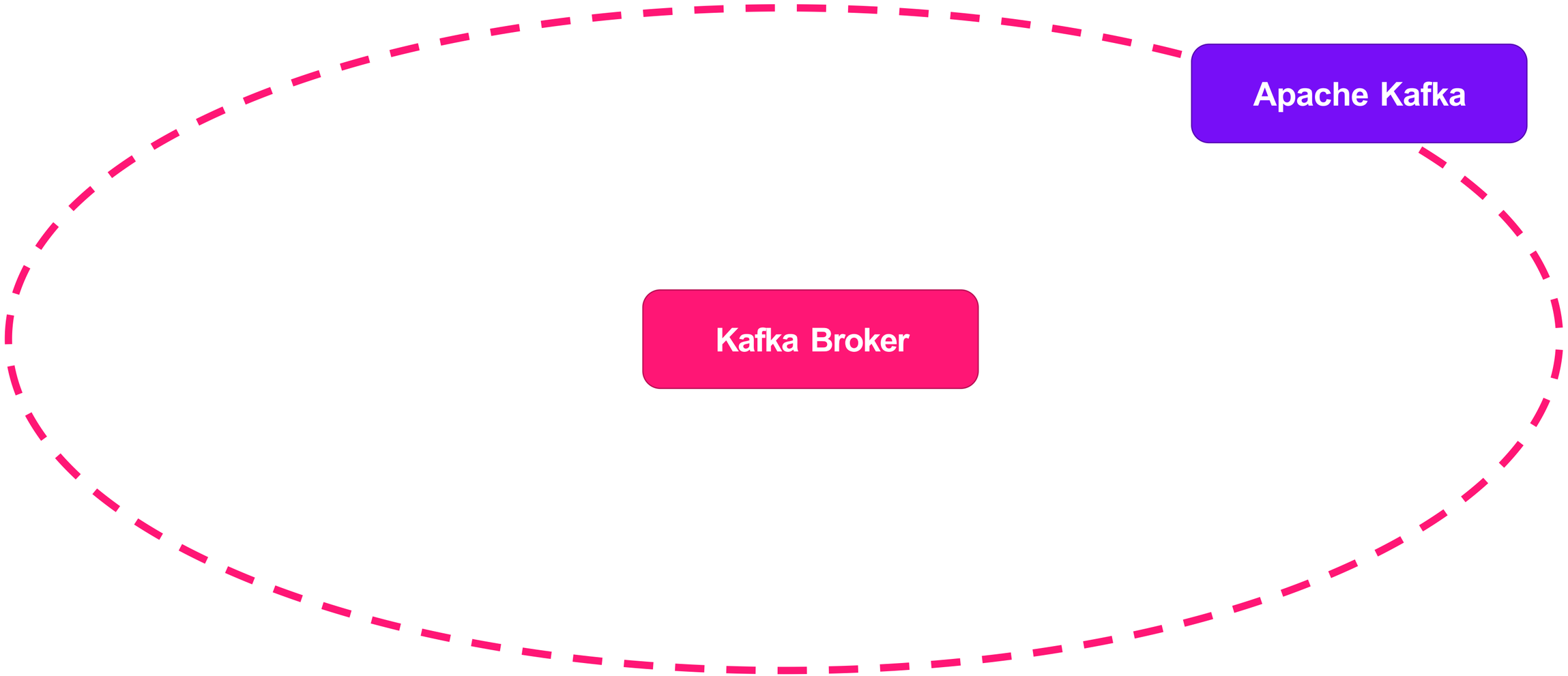
Kafka Streams

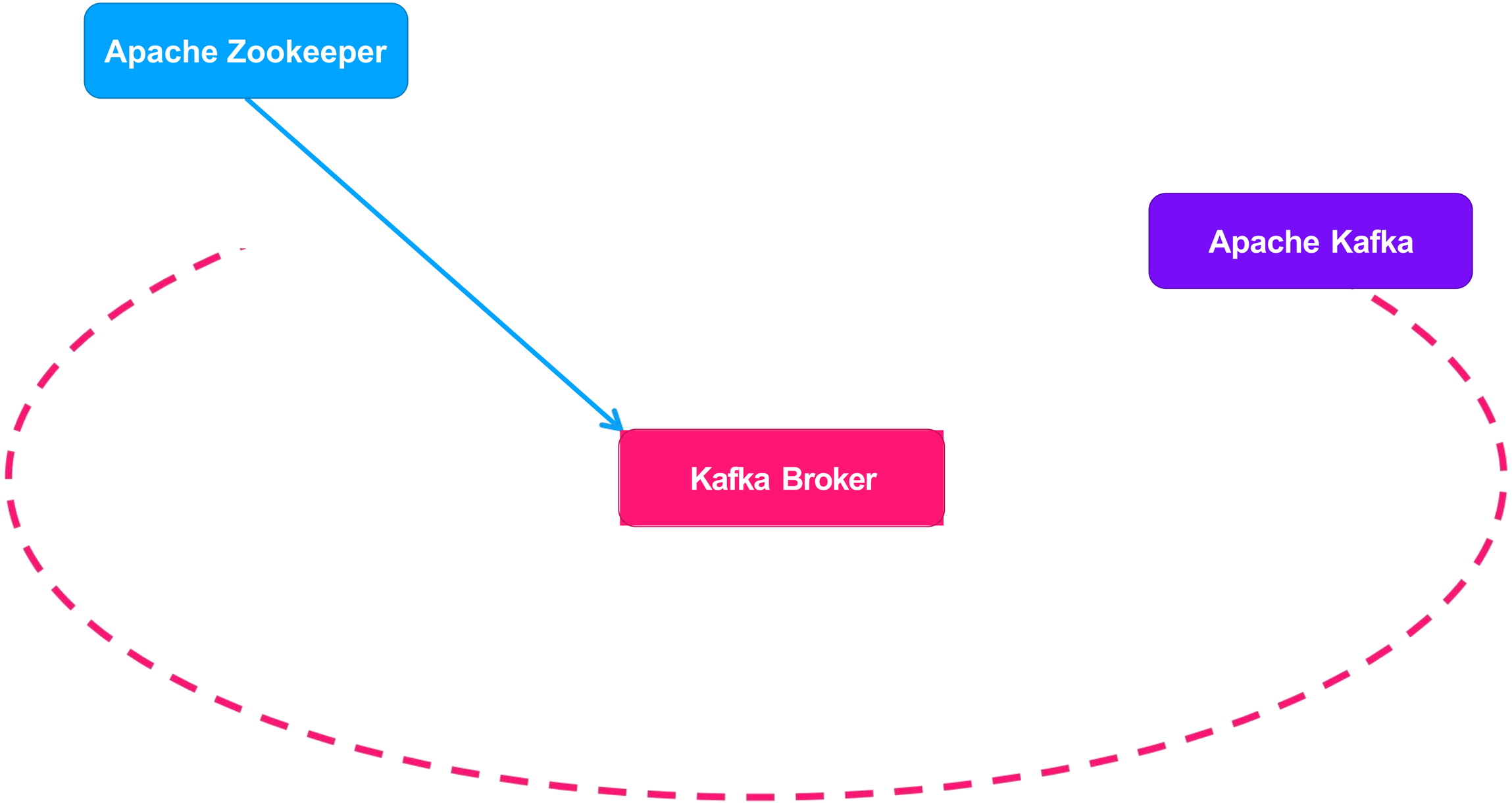
Perform administrative tasks

Creating and Configuring Producers and Consumers

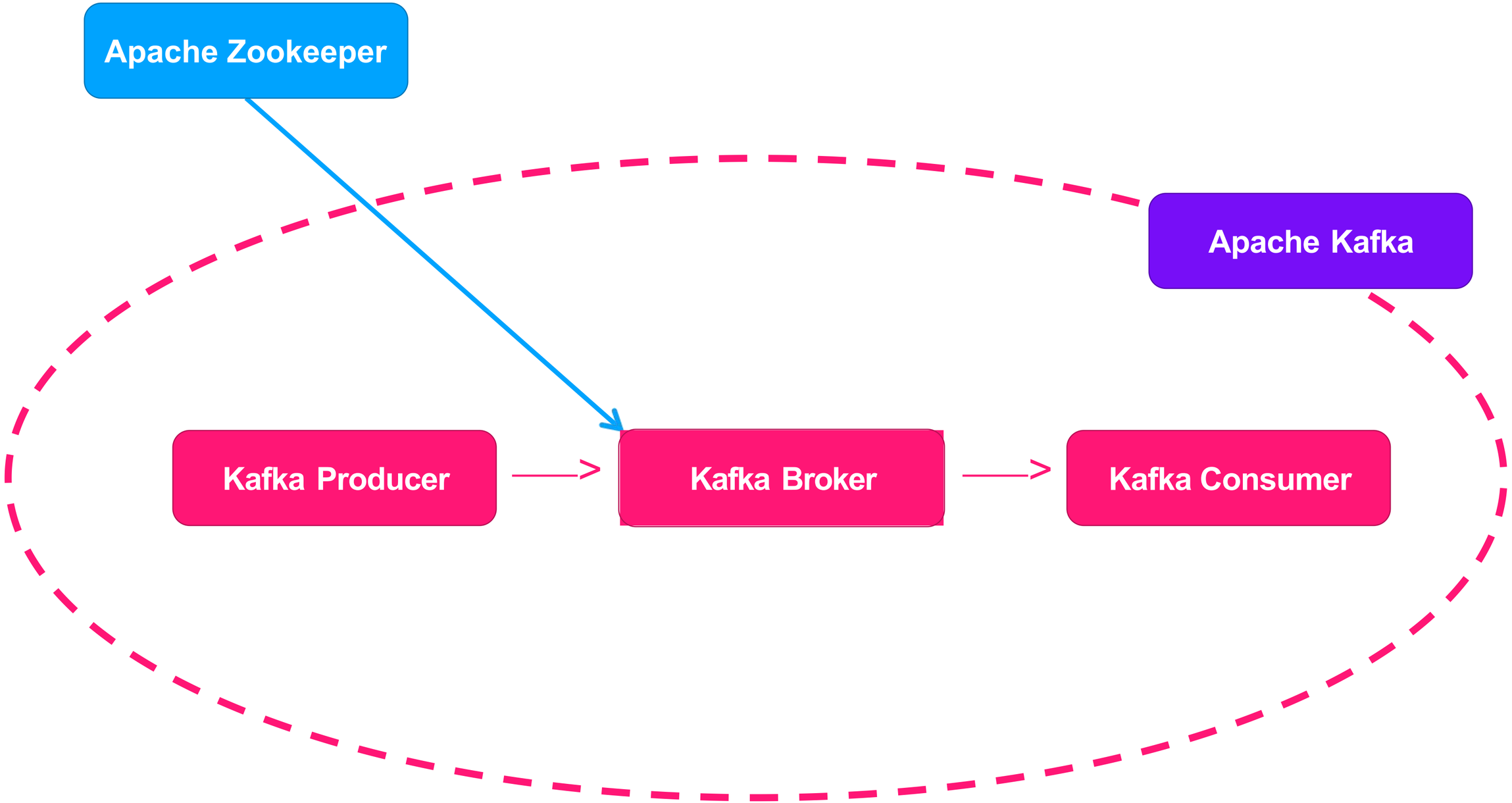
Kafka Connect

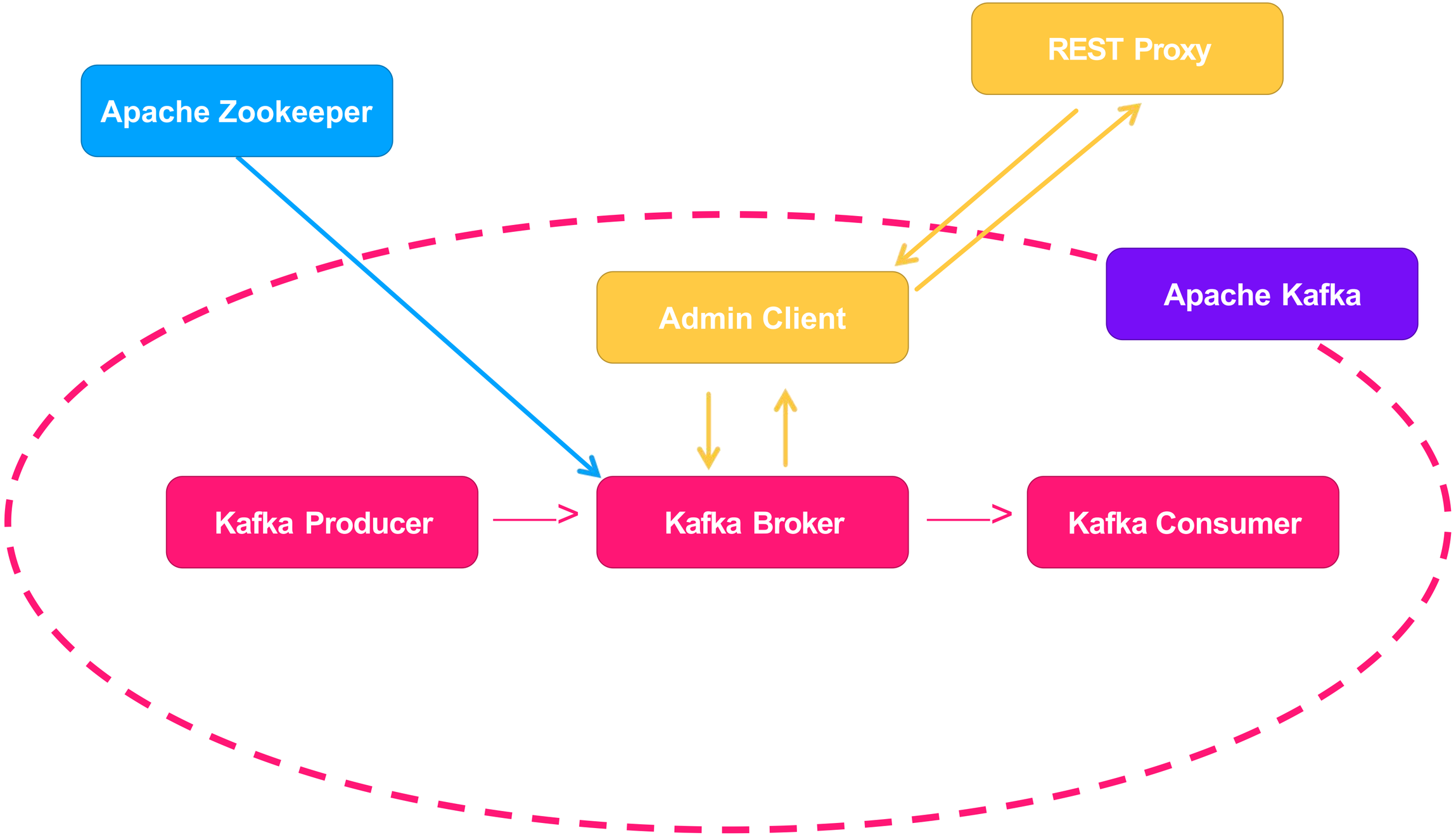
Meet Kafka

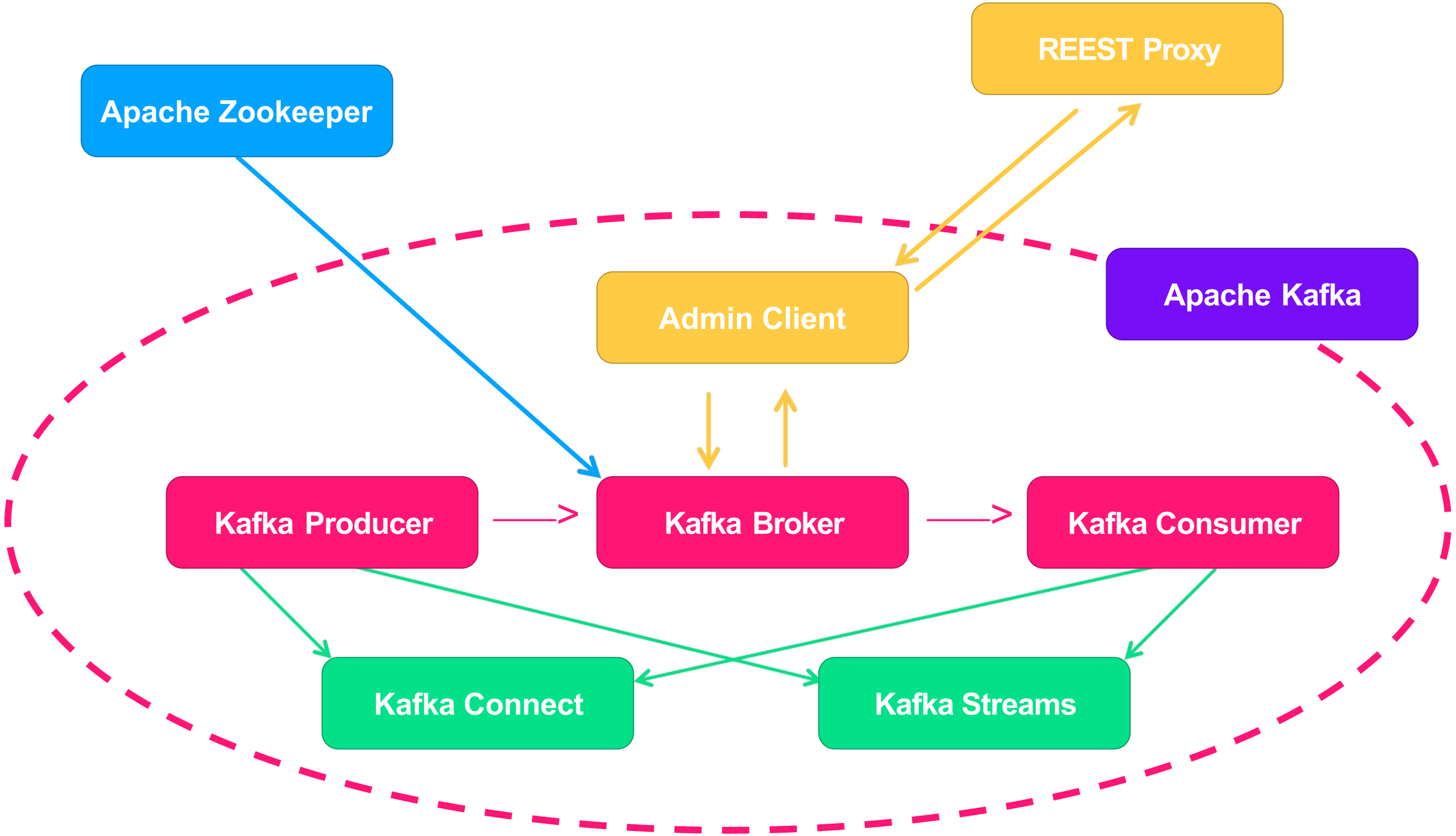

Apache Kafka
Kafka Broker

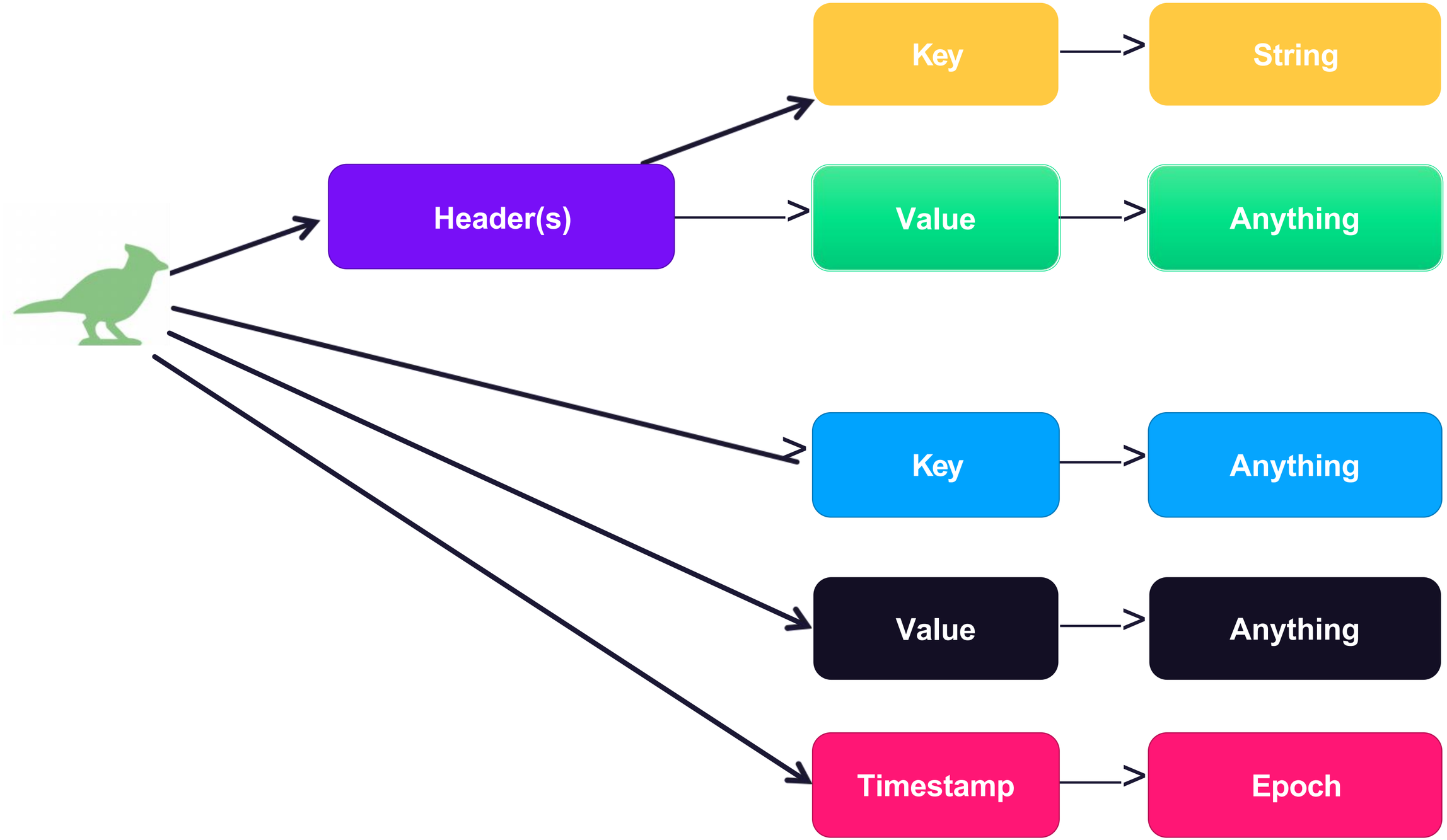

Apache Zookeeper
Apache Kafka
Kafka Broker

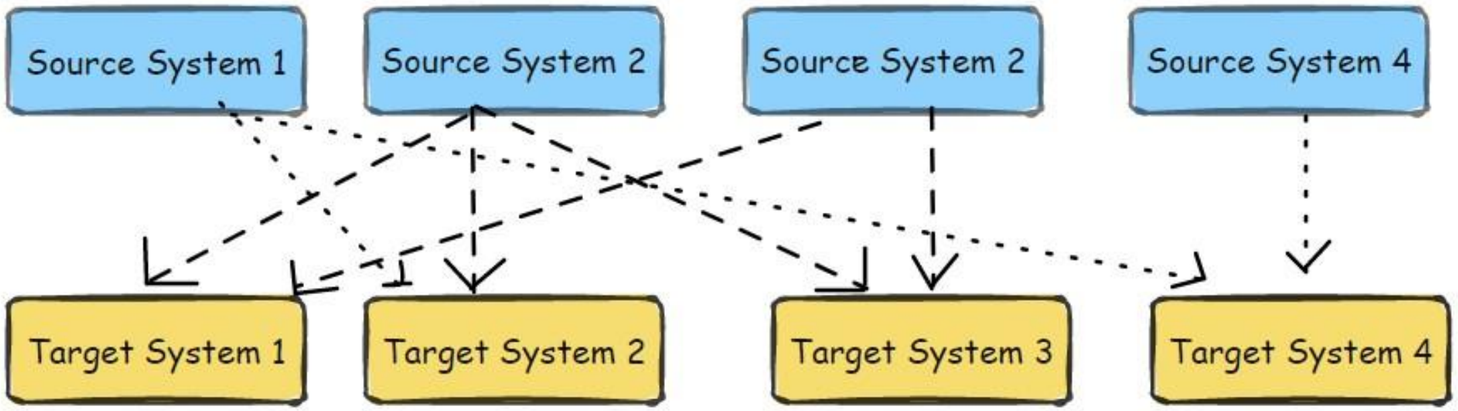
- Service Discovery
- Service Address Retrieval
- Management of Elections


Apache Zookeeper
Apache Kafka
Kafka Producer
Kafka Broker
Kafka Consumer


REST Proxy
Apache Zookeeper
Admin Client
Apache Kafka
Kafka Producer
Kafka Broker
Kafka Consumer


REEST Proxy
Apache Zookeeper
Admin Client
Apache Kafka
Kafka Producer
Kafka Broker
Kafka Consumer
Kafka Connect
Kafka Streams


Key
String
Header(s)
Value
Anything
Key
Anything
Value
Anything
Timestamp
Epoch



After a while it becomes complicated

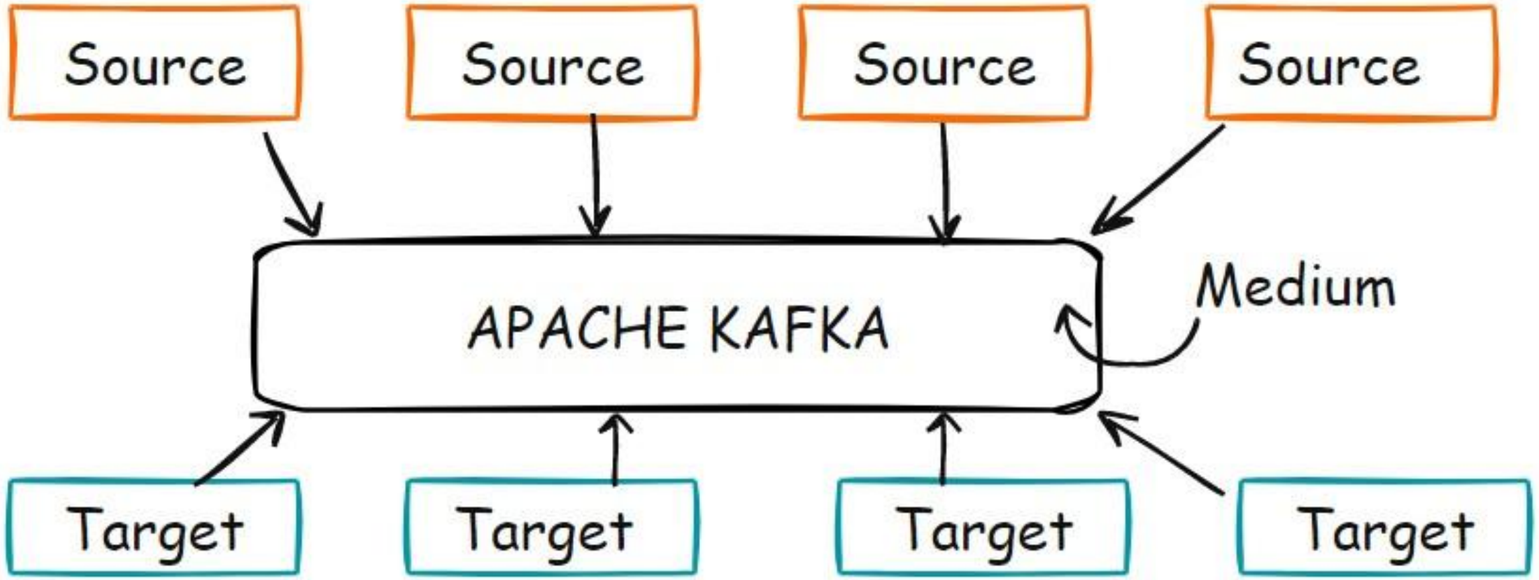
- 4 source system X 6 targets system
- 4 X 6 integrations

Challenges we will face

- Protocol – TCP/HTTP/JDBC/REST [How data is transport]
- Data Format – Data is passed in which [Binary , CSV, Array]
- Schema – How the data is shaped
- One more source is introduced , it will increase the network load
- Source & Target system have a very tight coupling

- Apache Kafka – Decoupling of data stream

- pub-sub model -: pub- Publisher , sub-Subscribes
- Pattern where a publisher sends a piece of data – not directly to the receiver, in fact it will send it through a medium



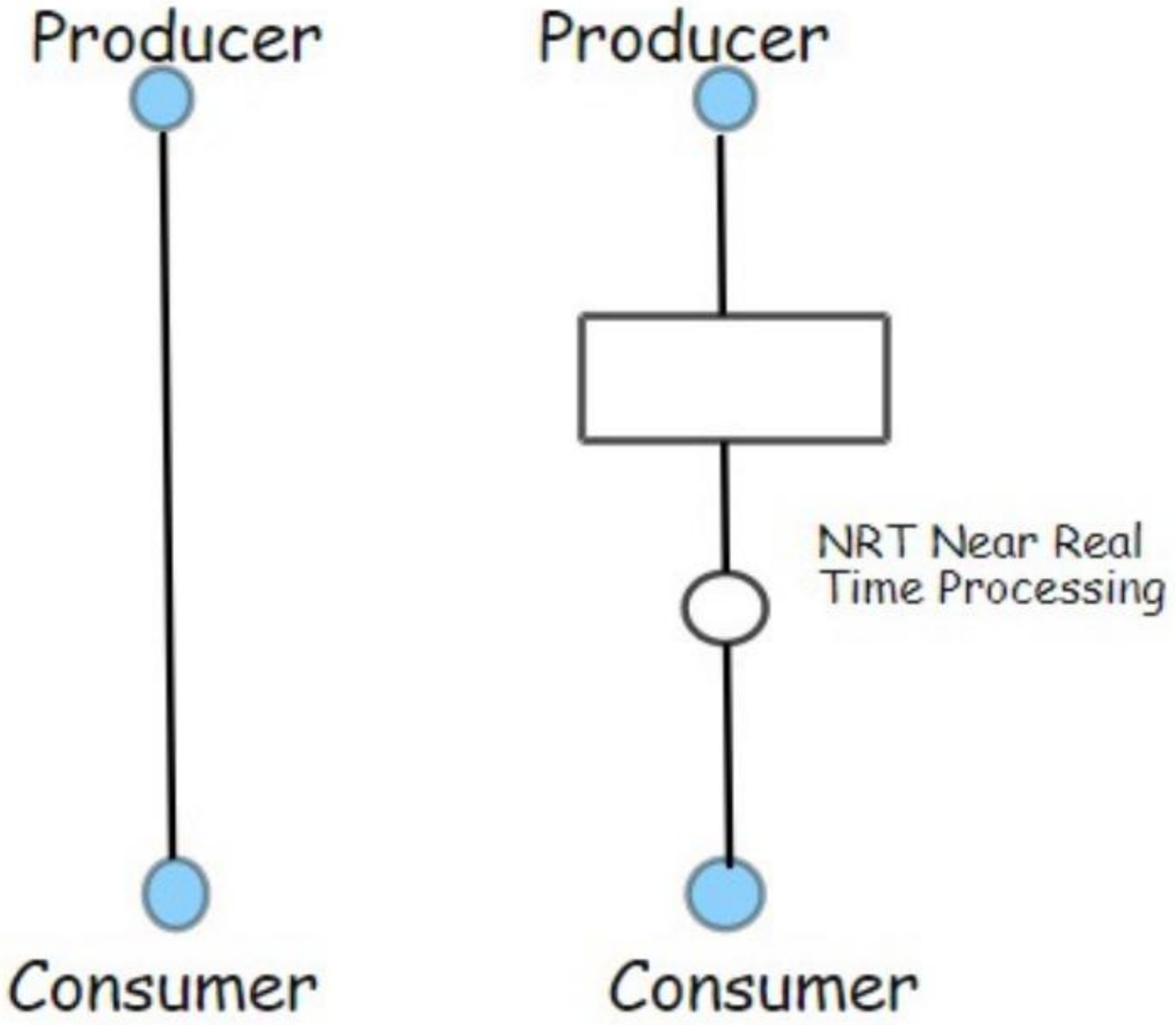
Why Kafka ?

- Created at LinkedIn , -- open source project– Distributed, Resilient , Fault Tolerant Confluent
- Horizontal Scaling
- 100’s of brokers , can scale to millions of messages per second
- High Performance
- Latency of less than 10 ms real time

- Apache Kafka – Use Cases
- Messaging System
- Activity Tracking
- Stream Processing
- Gather metric from different location
- Gather application logs
- Integrated with big data modules

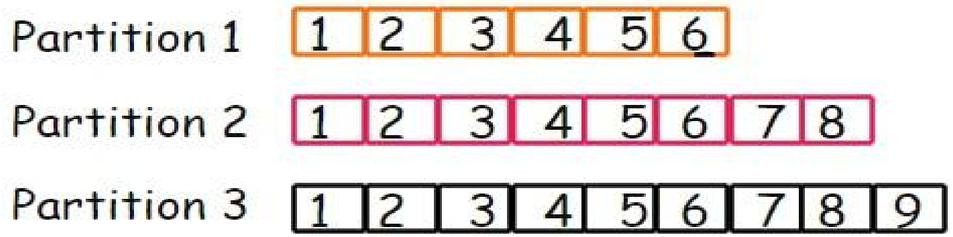
3 big differences which set Kafka aside

1. Distributed System – rather than running dozens of individual messaging brokers; hard wired to different apps. We have a central platform that can scale
2. Kafka is a true storage system – it is built to store the data as long as you want (default– 1 week) , Real Delivery Guarantee— [replicated | persisted]
3. Kafka supports Stream Processing →Other Messaging System → just hand-out the messages , With Stream Processing in Kafka – allows us to compute derived stream & generate more meaningful data


Producer
Consumer
Producer
NRT Near Real
Time Processing
Consumer

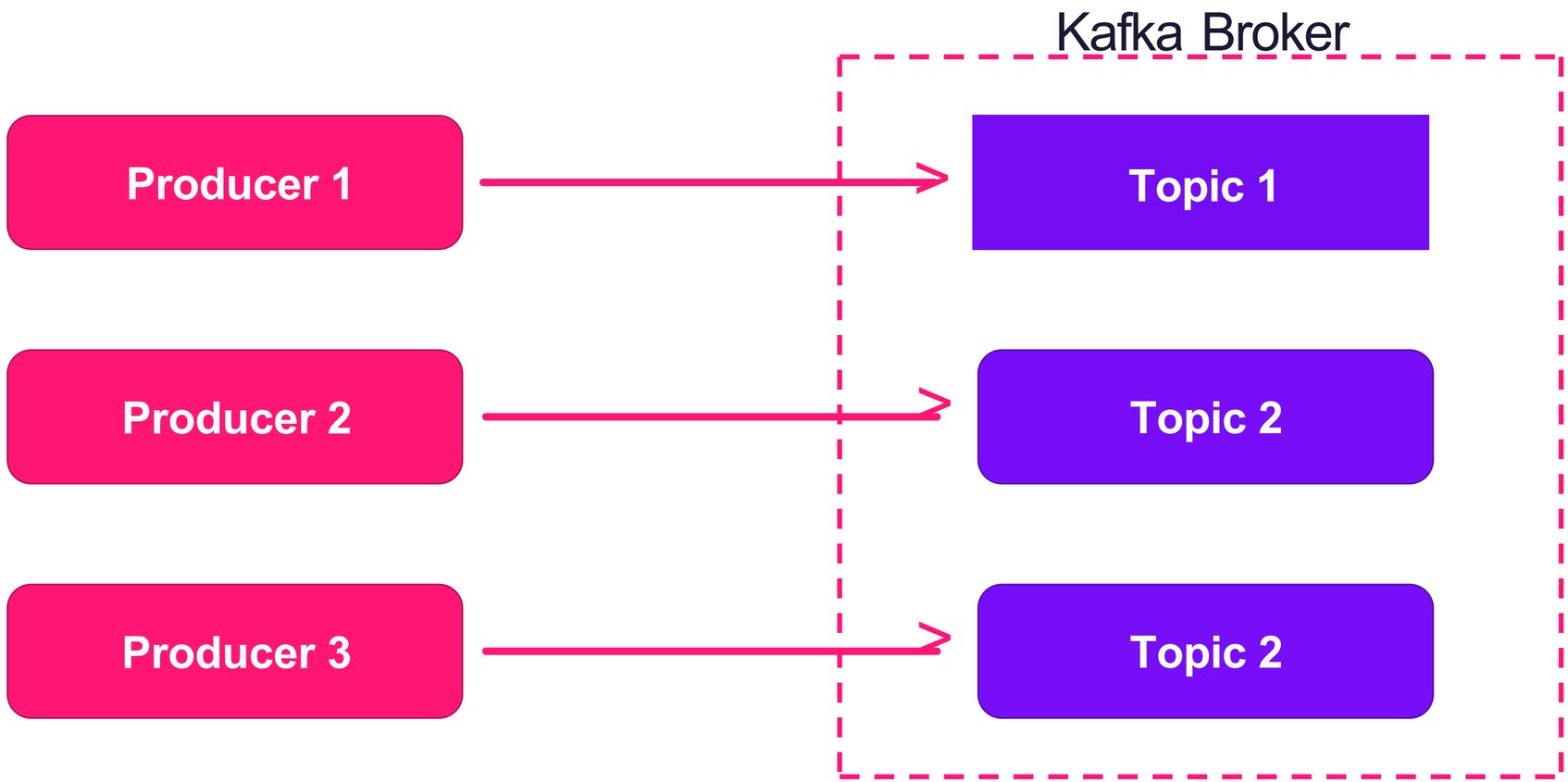
Topics , Partitions , Offset

- Topics- A particular stream of data similar to a table in a database (without all constraints)
- You can have as many as topics you want
- A topic is identified by its name
- Topics are split into Partitions (Kafka Topic)



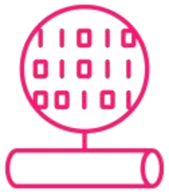
- Each partition is ordered
- Each message in a partition gets an incremental id , called as offset

Topics


Kafka Broker
Producer 1
Producer 2
Producer 3
Topic 1
Topic 2
Topic 2

Retention Can Be Set To:

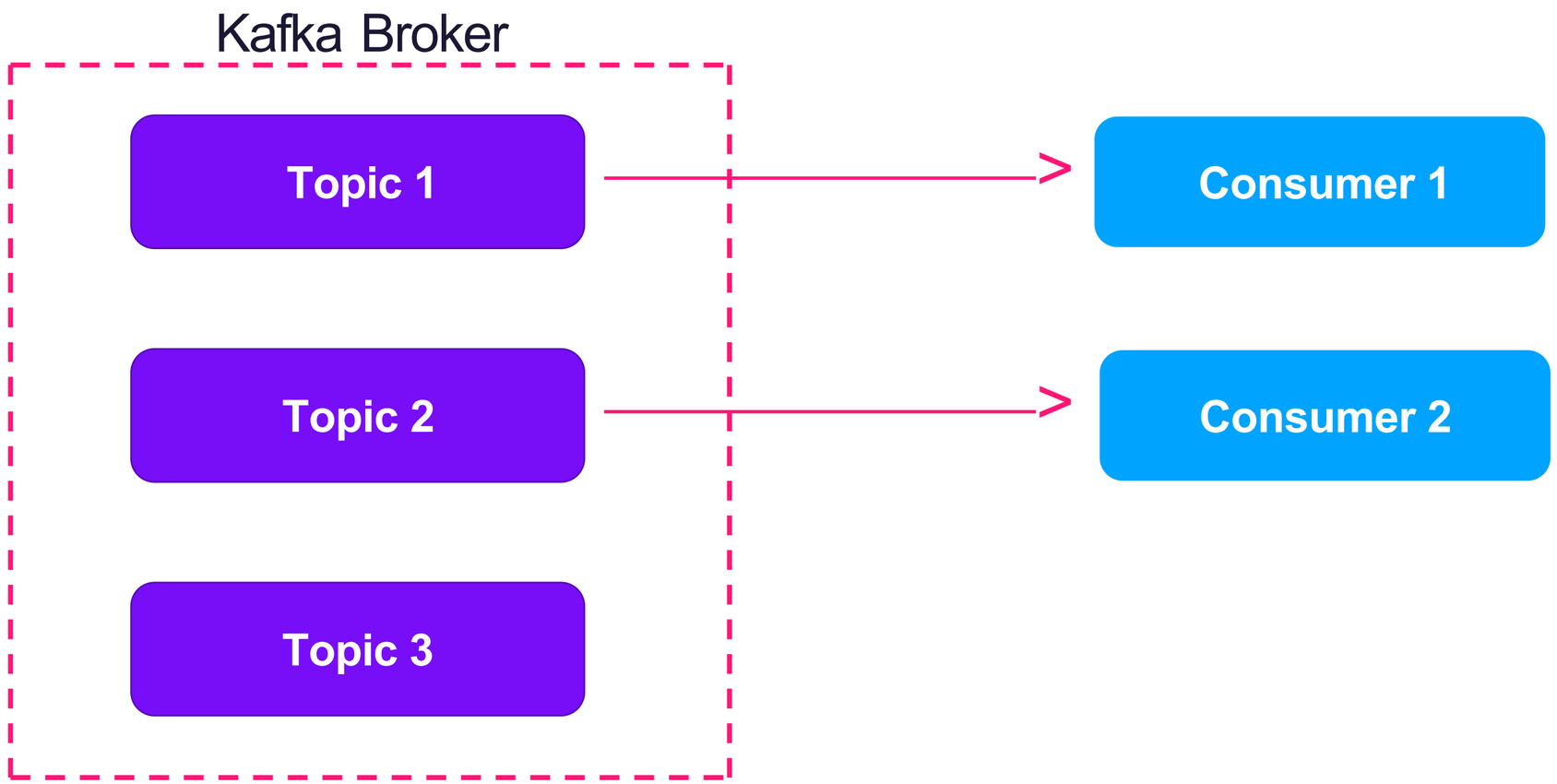
Days: Meaning only keeps messages of up to N days and then delete

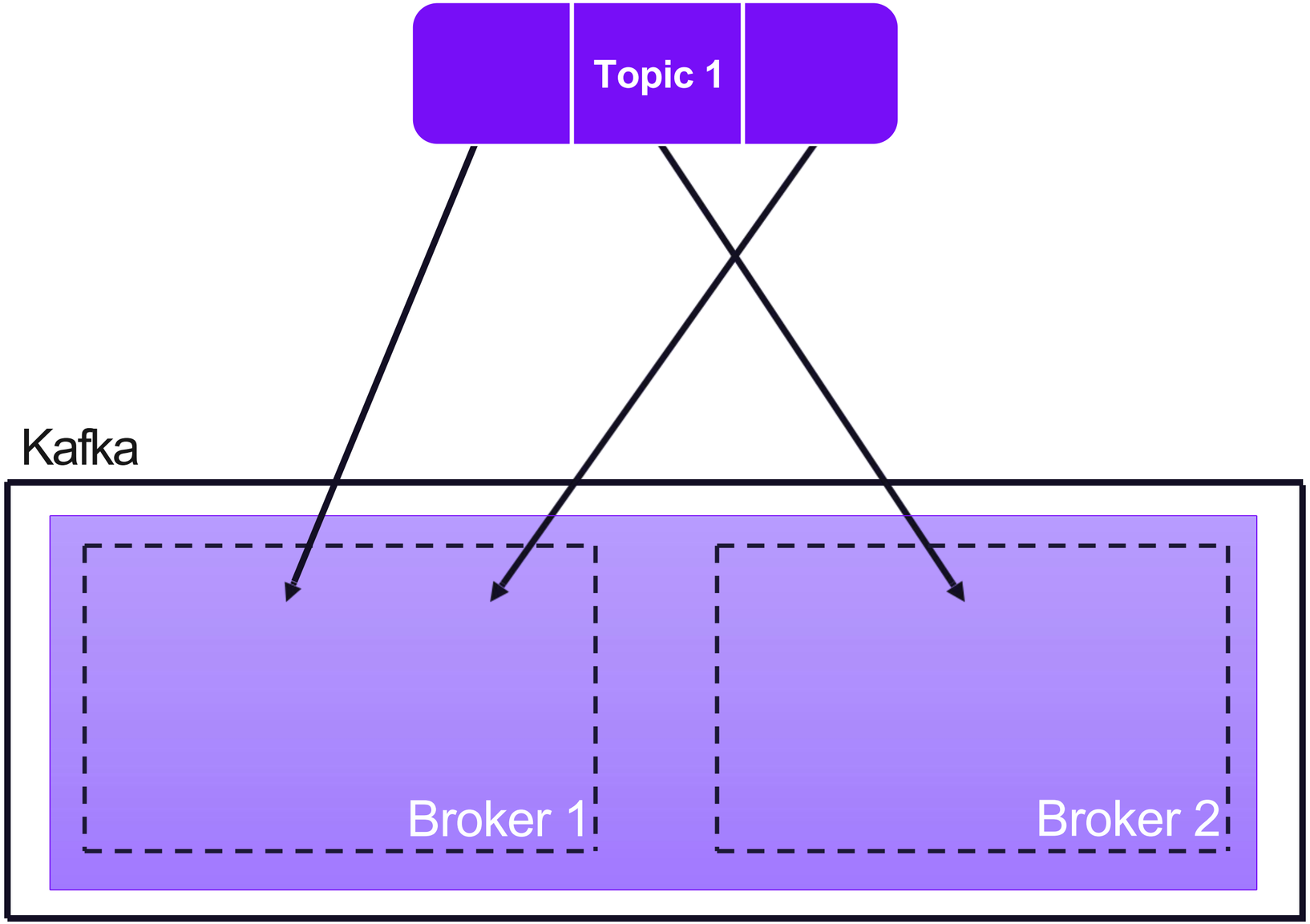


Bytes: Meaning only keep messages until the topic has more than X bytes and then delete

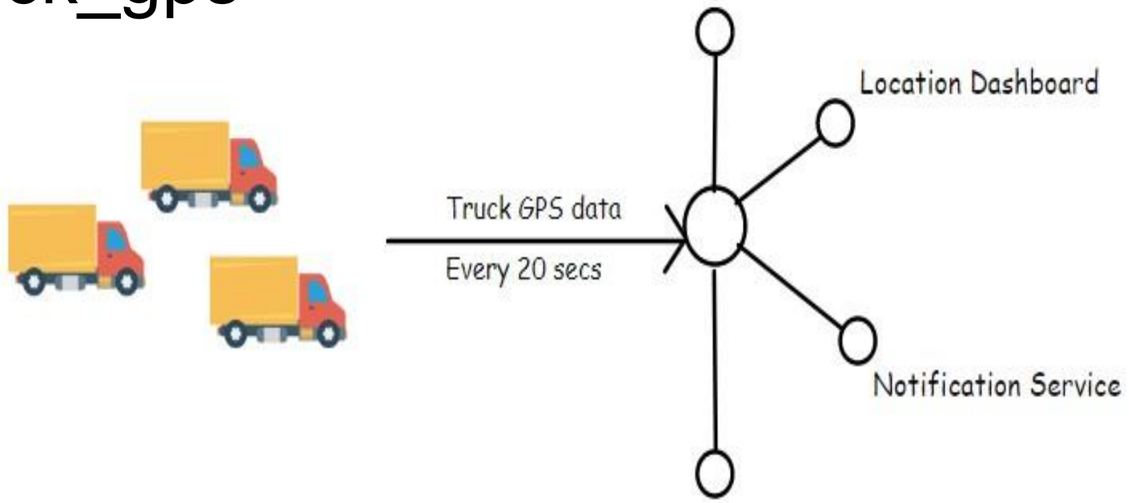
None: Meaning store all messages from the beginning of time

Topics



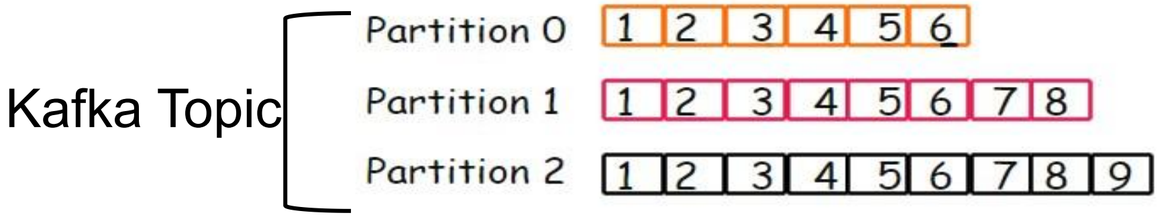

Topic 1
Kafka
Broker 1
Broker 2

Topic :-
Truck_gps



- Each truck sends the GPS position to Kafka
- Data is sent to the topic- truck_gps
- Each truck sends a message after 20 seconds each message contains truck_id & the truck position (latitude & longitude)

Offset



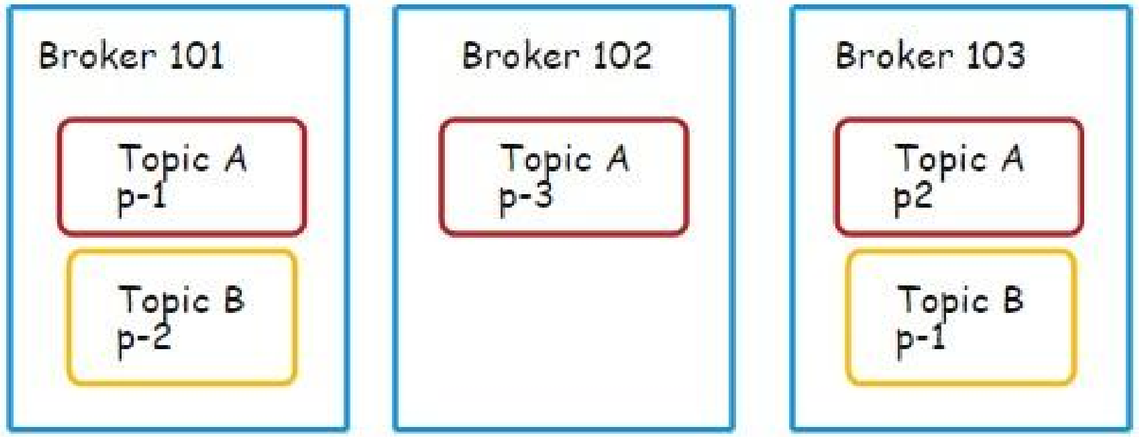
- Offsets have meaning only for a specific partition
- Order is guaranteed only within a partition (not across the partition)
- Data is kept for 1 week (configurable)
- Data once written to the partition it cannot be changed (immutable)
- Data is assigned to the partition randomly unless the key is provided

Brokers & Topics

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker is identified with its id(integer)
- Each broker contains certain topic partitions
- We connect to any broker– Bootstrap Server
- No of brokers-3

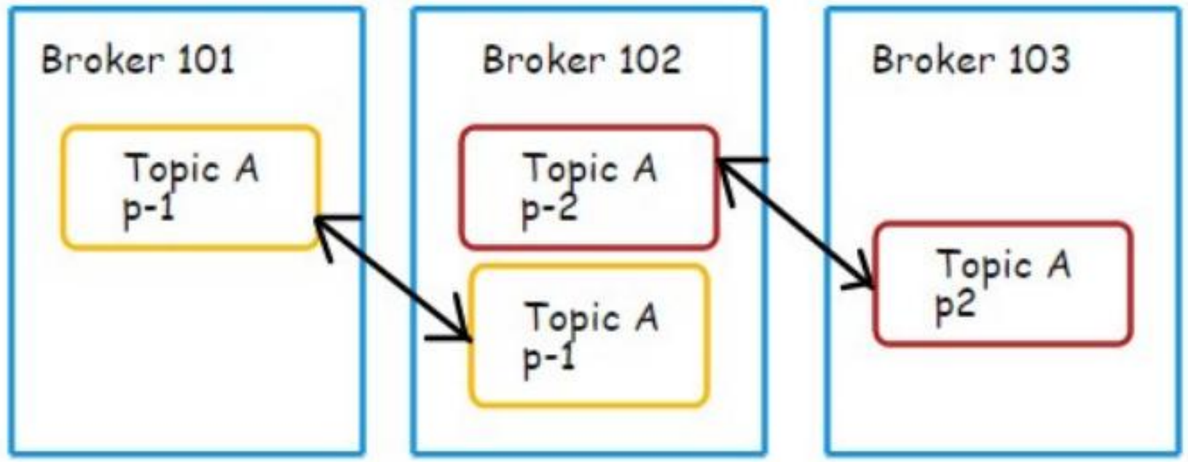
Topic A- 3 partitions

Topic B- 2 partitions



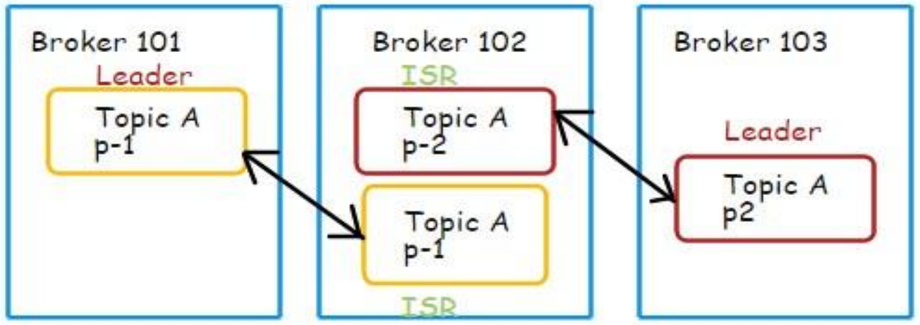
Topic Replication Factor

- A topic should have Replication Factor greater than 1 (usually between 2&3)
- Replication Factor make sure that if a broker is down ; another broker can serve the data
- Rf-2 [1 original + 1 copy]
- Topic A – 2 partition & RF- 2



Concept of Leader for a Partition

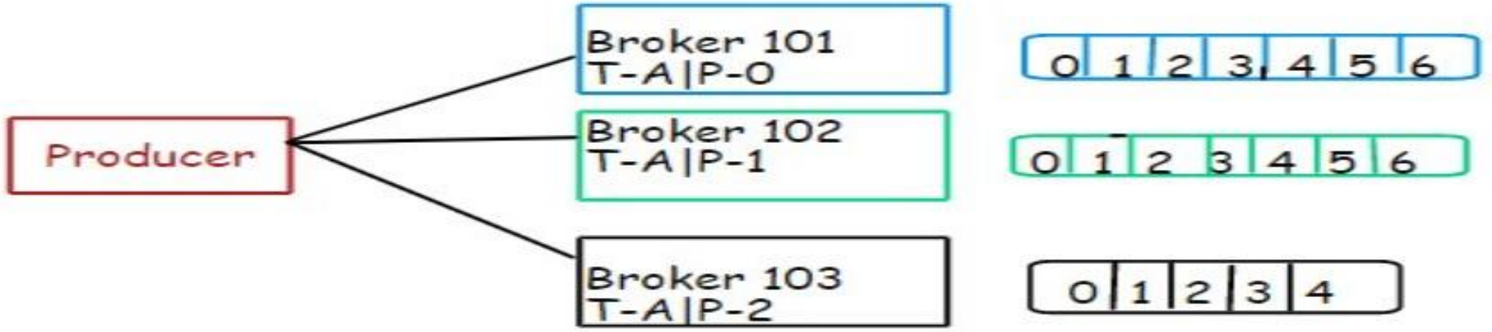
- [At any point in time; only one BROKER can be a LEADER for a PARTITION]
- [ONLY that LEADER can RECEIVE & SERVE DATA FOR that Partition]

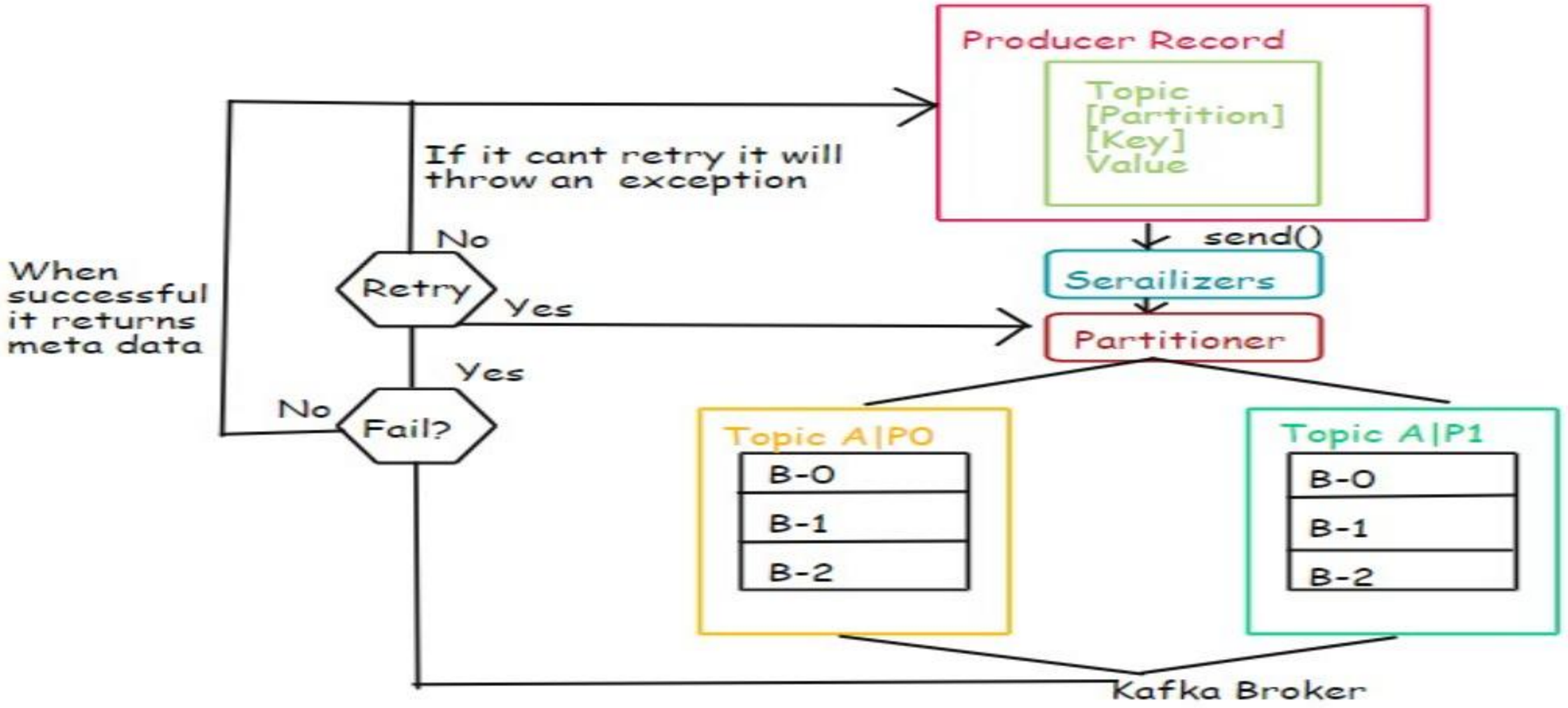


- Other brokers will synchronize the data
- Each partition has ONE LEADER & Multiple ISR[In Sync Replicas]

Producers

- Producer writes the data to the topics
- Producers automatically knows to which broker &partition to write to.
- In case a Broker fails; Producer will automatically recover


Producer
Broker 101
T-A|P-0
Broker 102
T-A|P-1
Broker 103
T-A|P-2
0 1 2 3 4 5 6
0 1 2 3 4 5 6
0 1 2 3 4


Producer Record
Topic
[Partition]
[Key]
Value
send()
Serailizers
Partitioner
If it cant retry it will
throw an exception
No
Retry
Yes
When
successful
it returns
meta data
Yes
No
Fail?
Topic A|P0
B-0
B-1
B-2
Topic A|P1
B-0
B-1
B-2
Kafka Broker

- Producers can choose to receive the acknowledgment of data writes
- Acks=0| Producer will not wait for acknowledgment

(possible data loss)

- Acks=1| Producer will wait for ‘Leaders Acknowledgment

(limited data loss)

- Acks=all | Leader + Replicas Acknowledgment (no data loss)

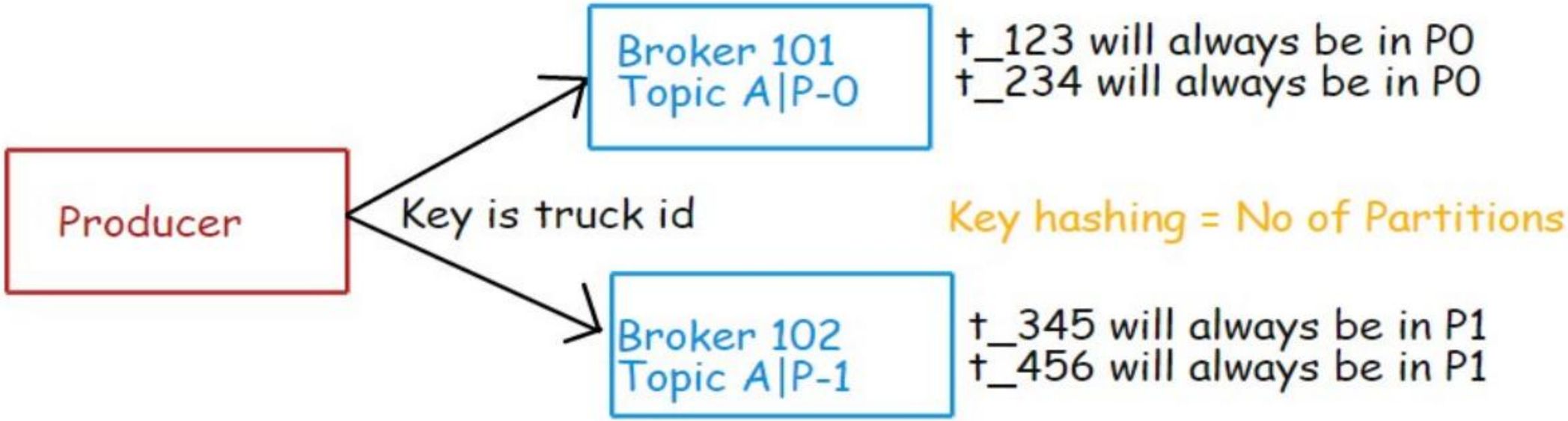
- Producer– Message Key’s

➢Producer can choose to send a key with the message (string , number, etc)

➢If key=null; data is sent in Round Robin (101-102-103)

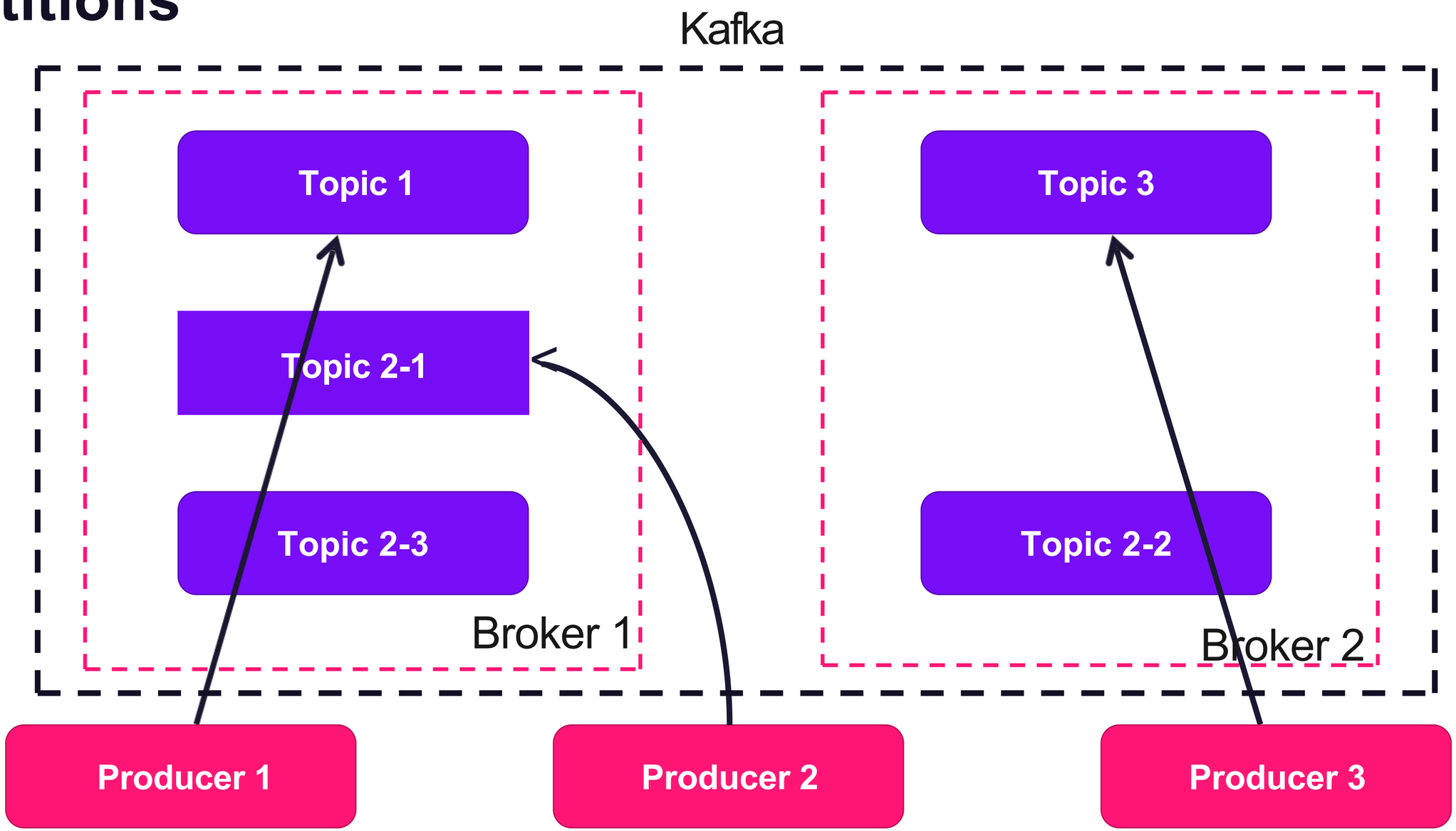
➢If key is sent ; then all the messages for that particular key will always land up in same partition

❖A key is normally sent with the message when we need ordering for a specific field ex:truck_id

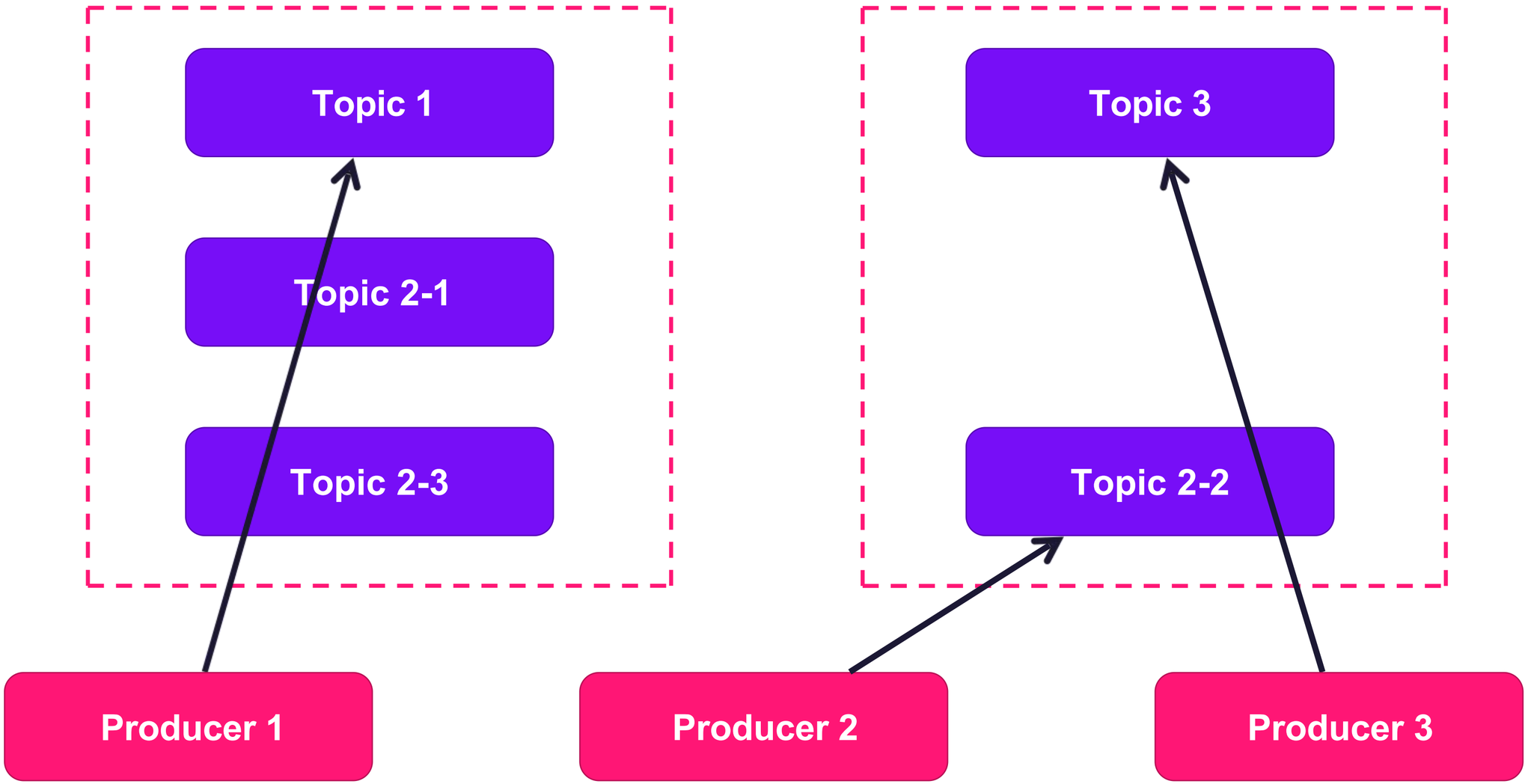

Broker 101
Topic A|P-0
t_123 will always be in P0
t_234 will always be in P0
Producer
Key is truck id
Key hashing = No of Partitions
Broker 102
Topic A|P-1
t_345 will always be in P1
t_456 will always be in P1

The partitioner uses the key to distribute the messages to the correct partition

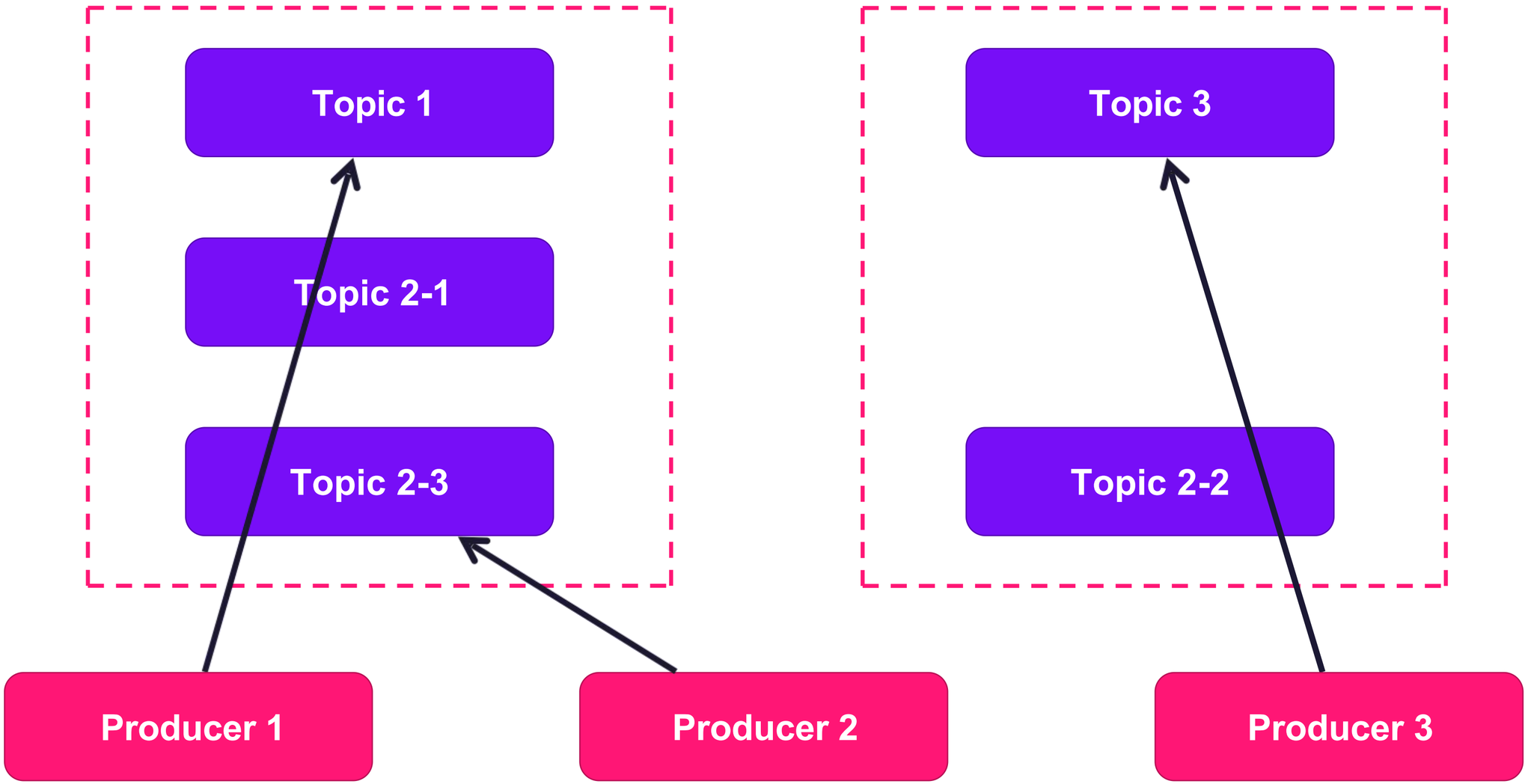
Partitions


Kafka
Topic 1
Topic 2-1
Topic 2-3
Broker 1
Topic 3
Topic 2-2
Broker 2
Producer 1
Producer 2
Producer 3

Partitions

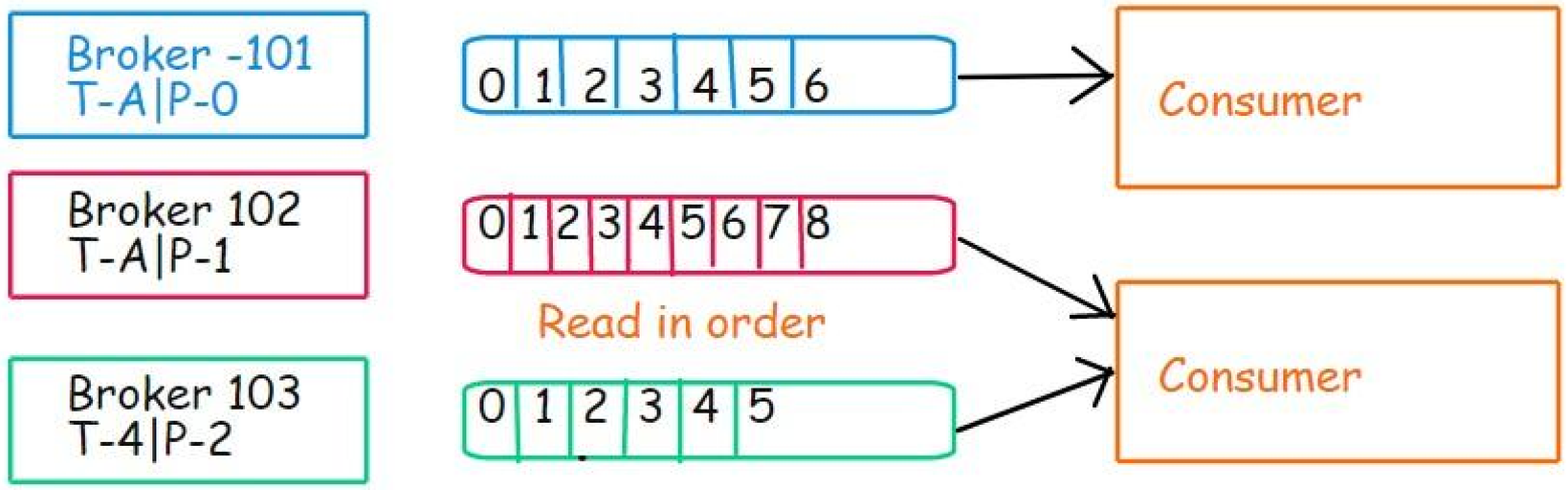

Topic 1
Topic 2-1
Topic 2-3
Topic 3
Topic 2-2
Producer 1
Producer 2
Producer 3

Partitions


Topic 1
Topic 2-1
Topic 2-3
Topic 3
Topic 2-2
Producer 1
Producer 2
Producer 3

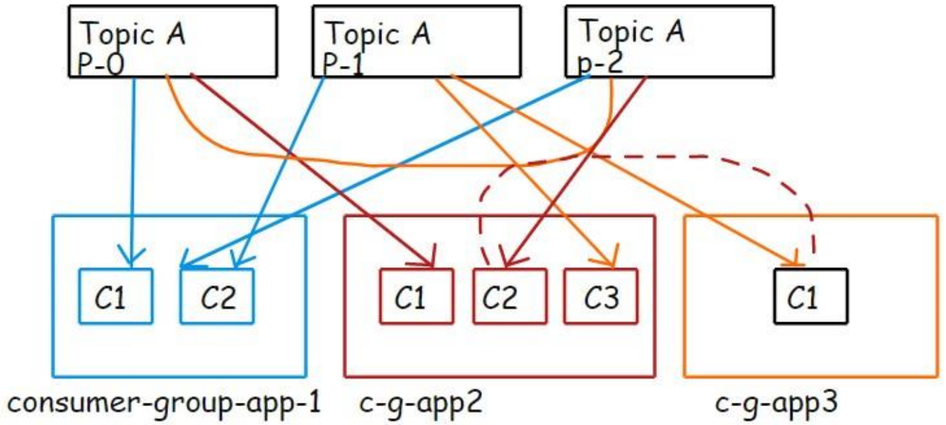
Consumer

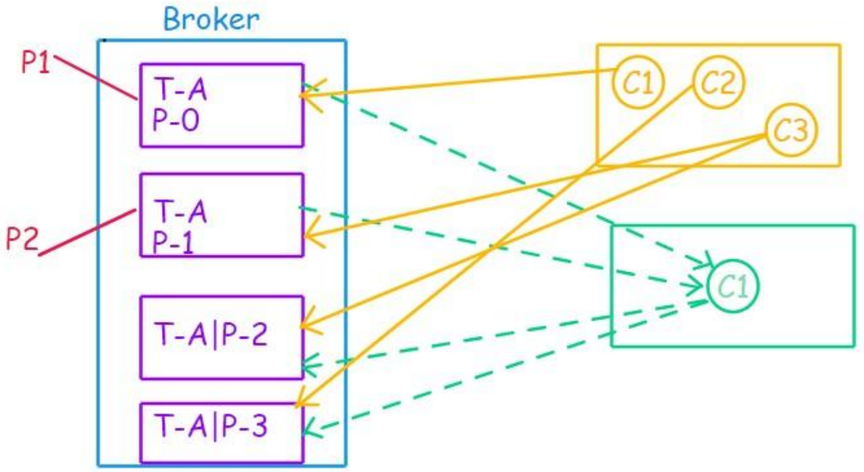
- Consumers read the data from a topic
- Consumer knows which broker to read from
- In case of broker failure ; consumer knows how to recover
- Data is read from each partition in an order



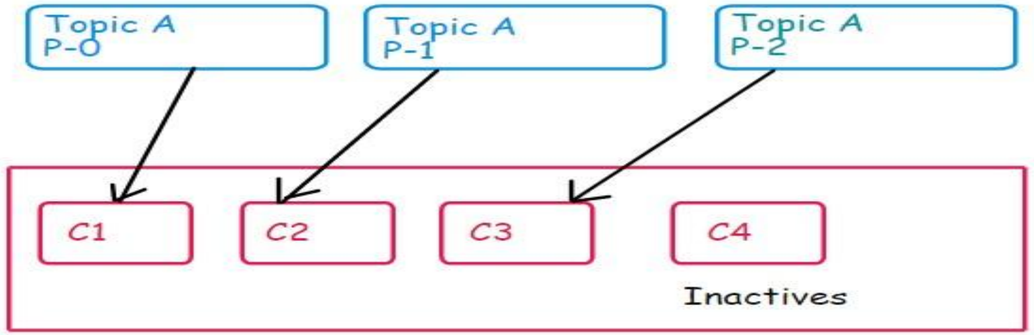
Consumer Group

- Consumer reads the data in Consumer group
- Each customers within a group reads from a exclusive partition
- If you have more consumers than the partitions; some consumers will be inactive
- Consumer belonging to the same group shares a group_id
- Consumer in the group divides the topics partition fairly amongst themselves
  - Each partition is only consumed by a single consumer in that group





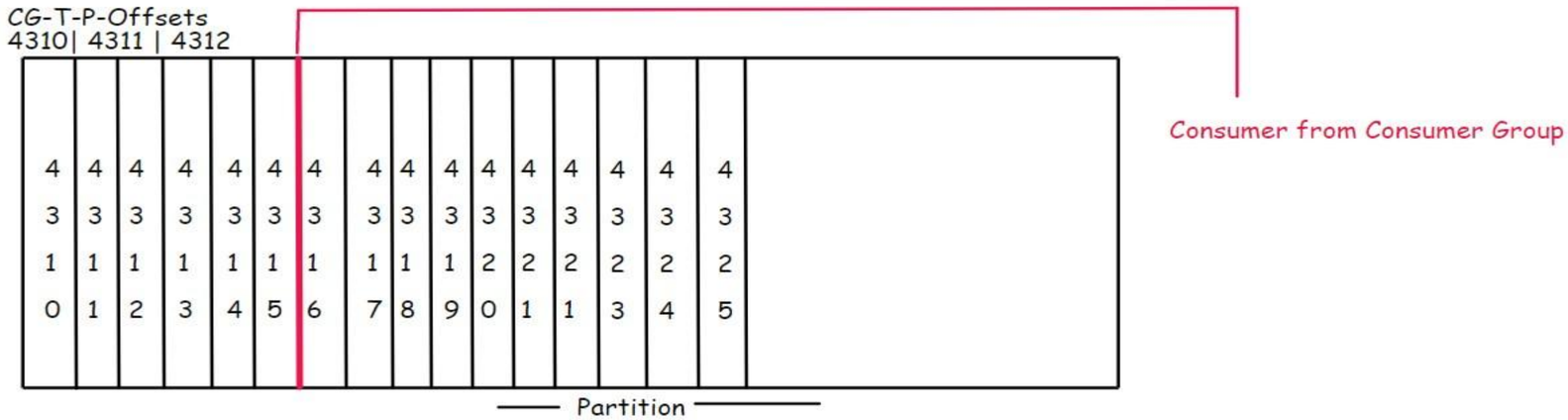
- What will happen if number of consumers are more than number of partitions?



- Rebalancing :- When a consumer group scales up or scales down Consumers in the group split the partition amongst themselves
- Rebalancing is triggered by a shift in ownership between the partition & consumer
- CG→ (C1 , C2 ,C3)→C2 (suffers a failure C1 & C3 will briefly pause consumption of messages from their partitions all the partitions will be up for reassignment

Consumer Offsets

- Kafka stores the offset at which consumer group has been reading
- The offset committed → they are stored in a Kafka topic _ __consumer__offsets
- When a Consumer group have processed the data received from Kafka : it commits the offsets

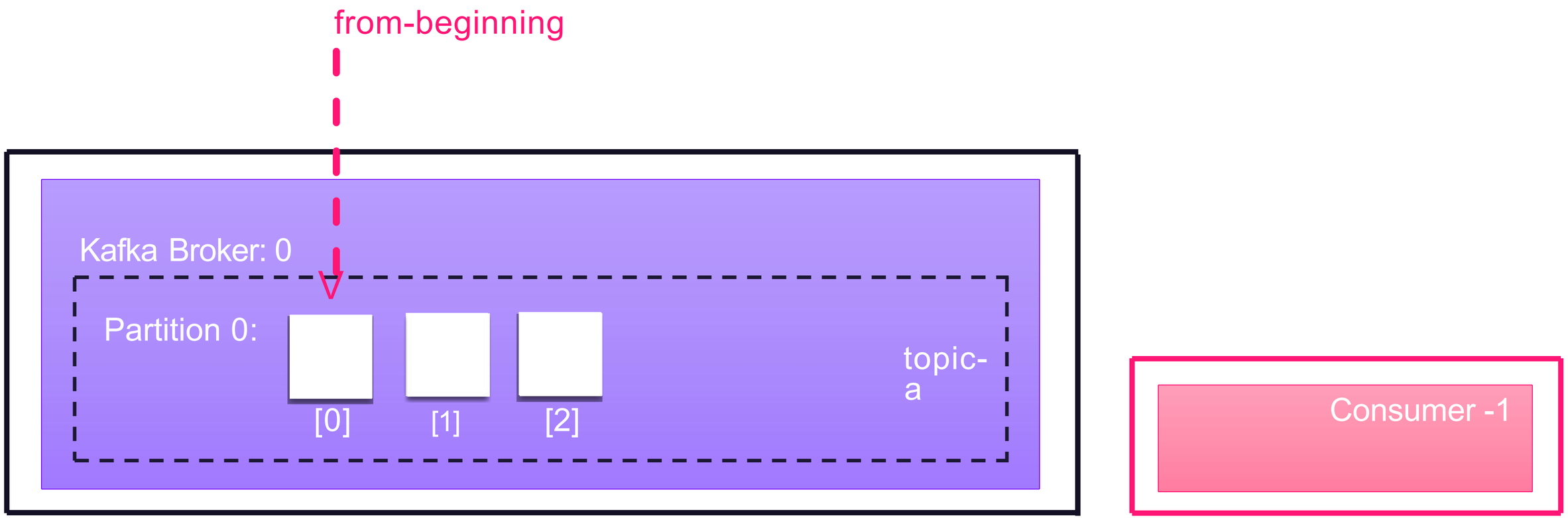


- If a consumer is down; it will be able to read from where it left

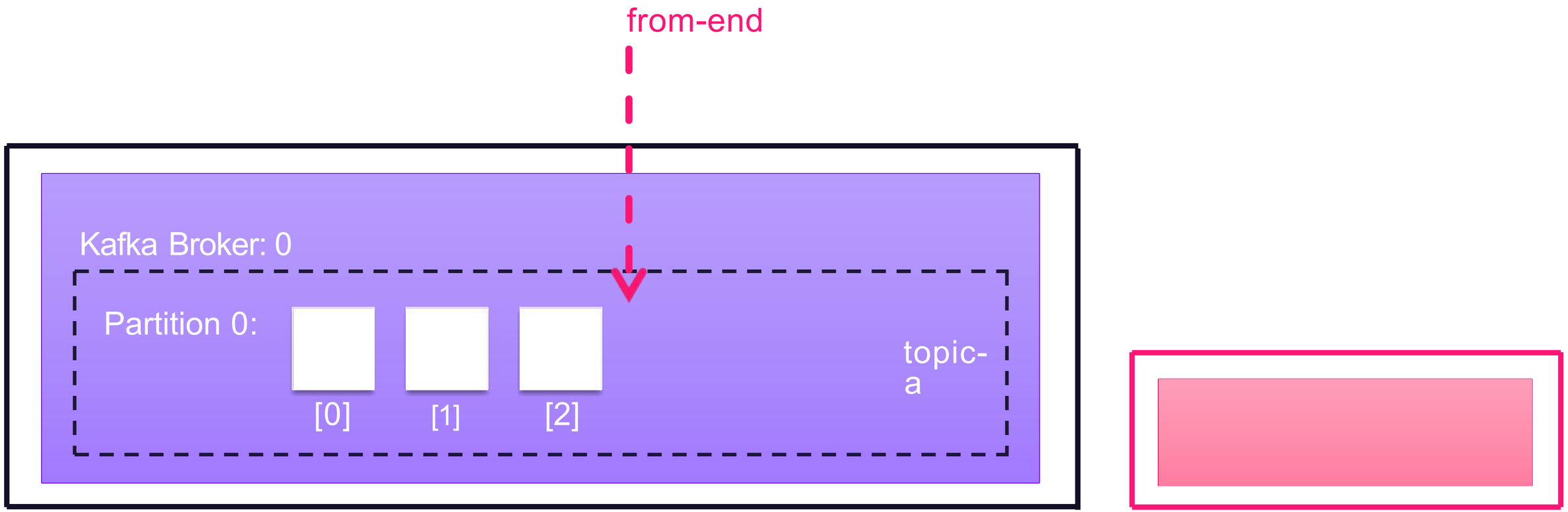
Delivery Semantics for Consumer

- Consumer can choose when to commit offset
- There are 3 delivery semantics
- At most once
  - ❖ Offsets is committed as soon as the message is received
  - ❖ If processing goes wrong; message will be lost
- At least once (usually preferred)
  - ❖ Offset is committed after the message is processed
  - ❖ If the processing goes wrong; message will be read again
  - ❖ Idempotent (if process is done multiple time it doesn't affect the state of application)

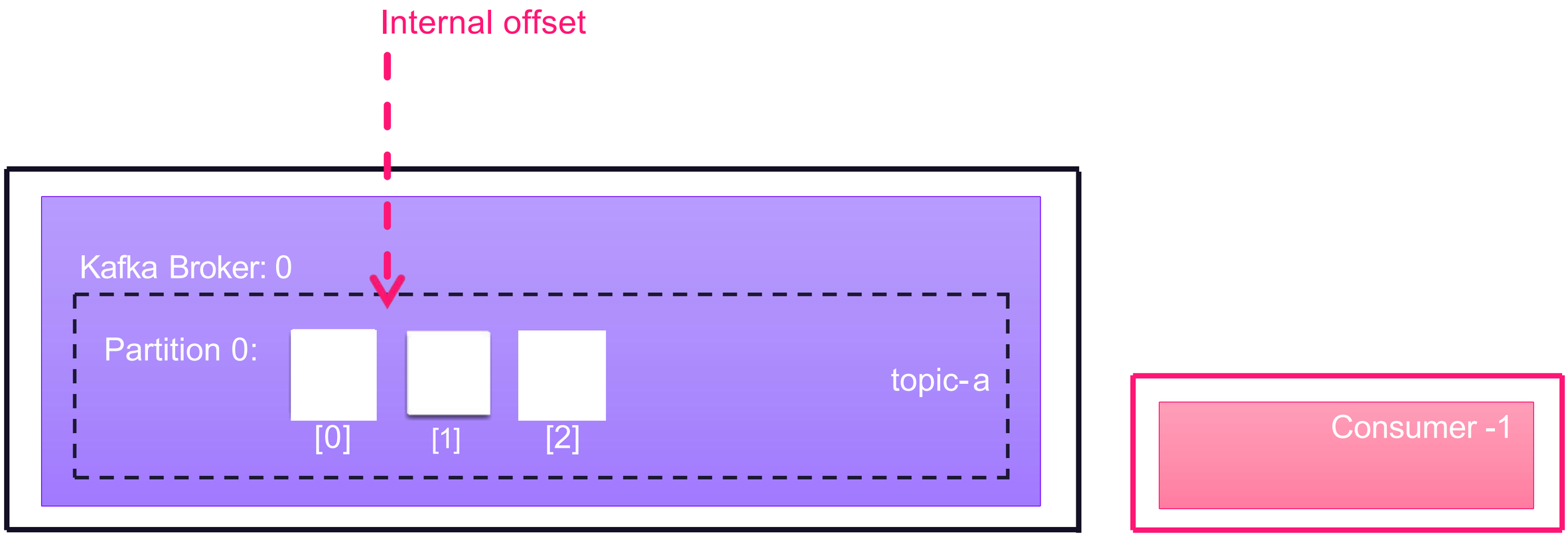
Consumer Offset: From Beginning



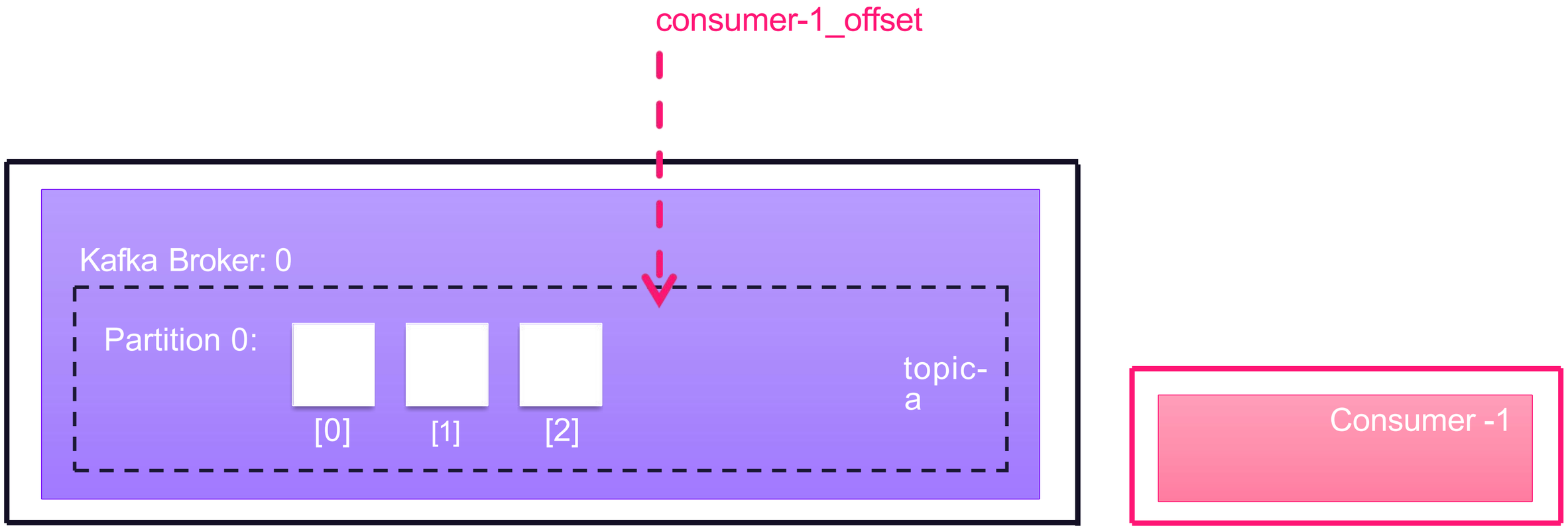
Consumer Offset: From End



Consumer Offset: Already Have Been Here

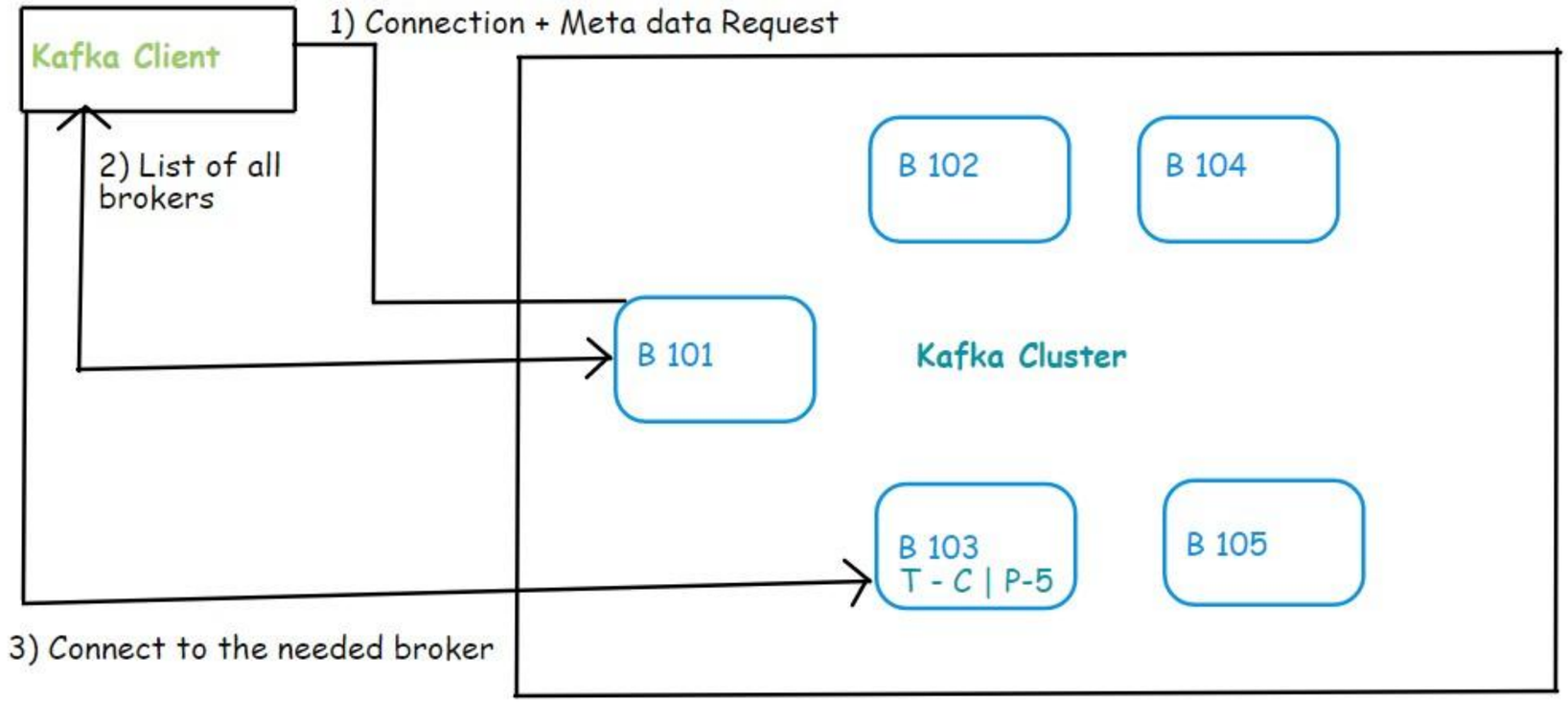


Consumer Offset



Kafka Broker Discovery

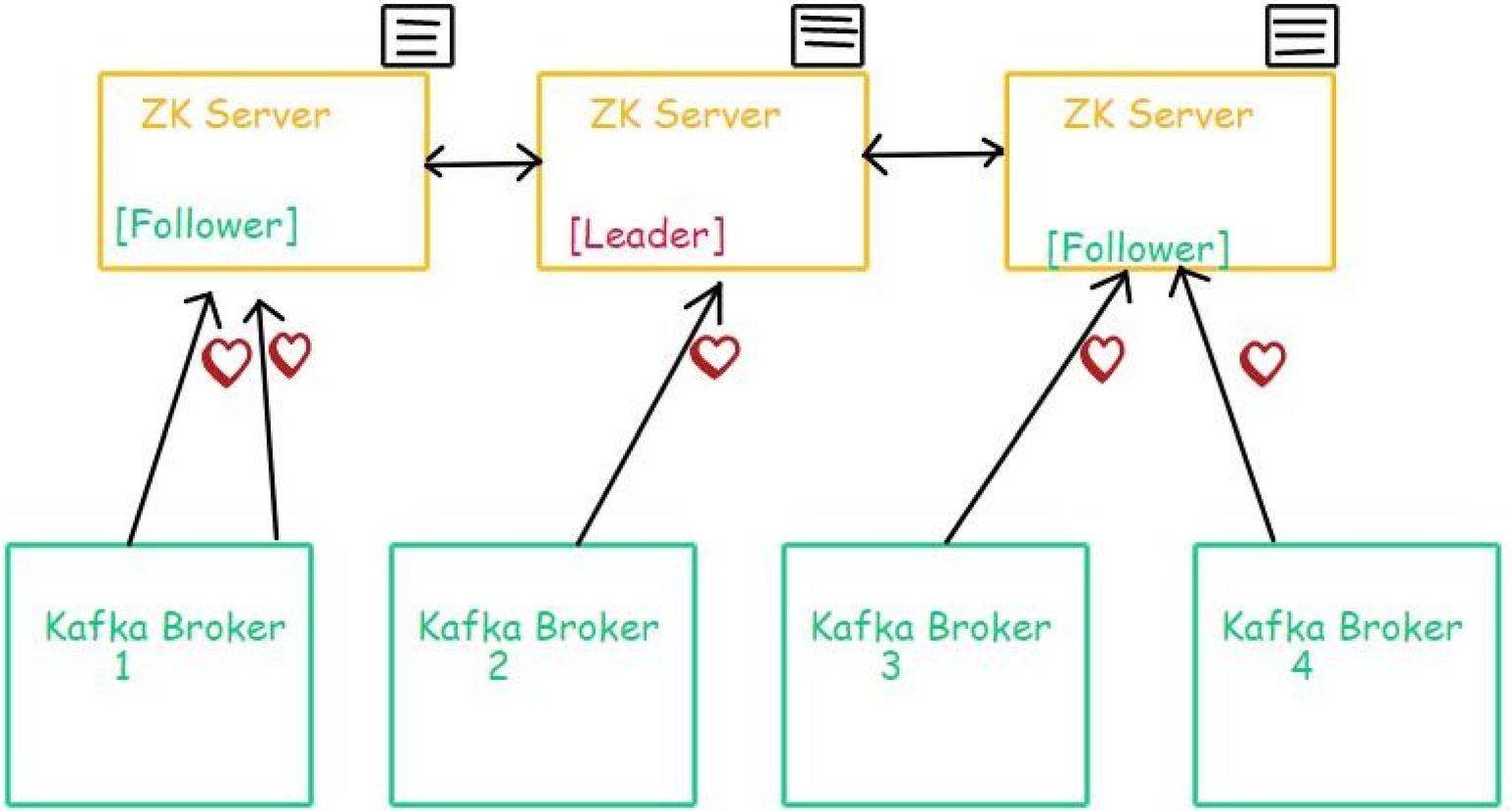
- Every Kafka Broker is called as “Bootstrap Server ”
- You need to connect to any one broker & you will be connected to entire cluster



Zookeeper

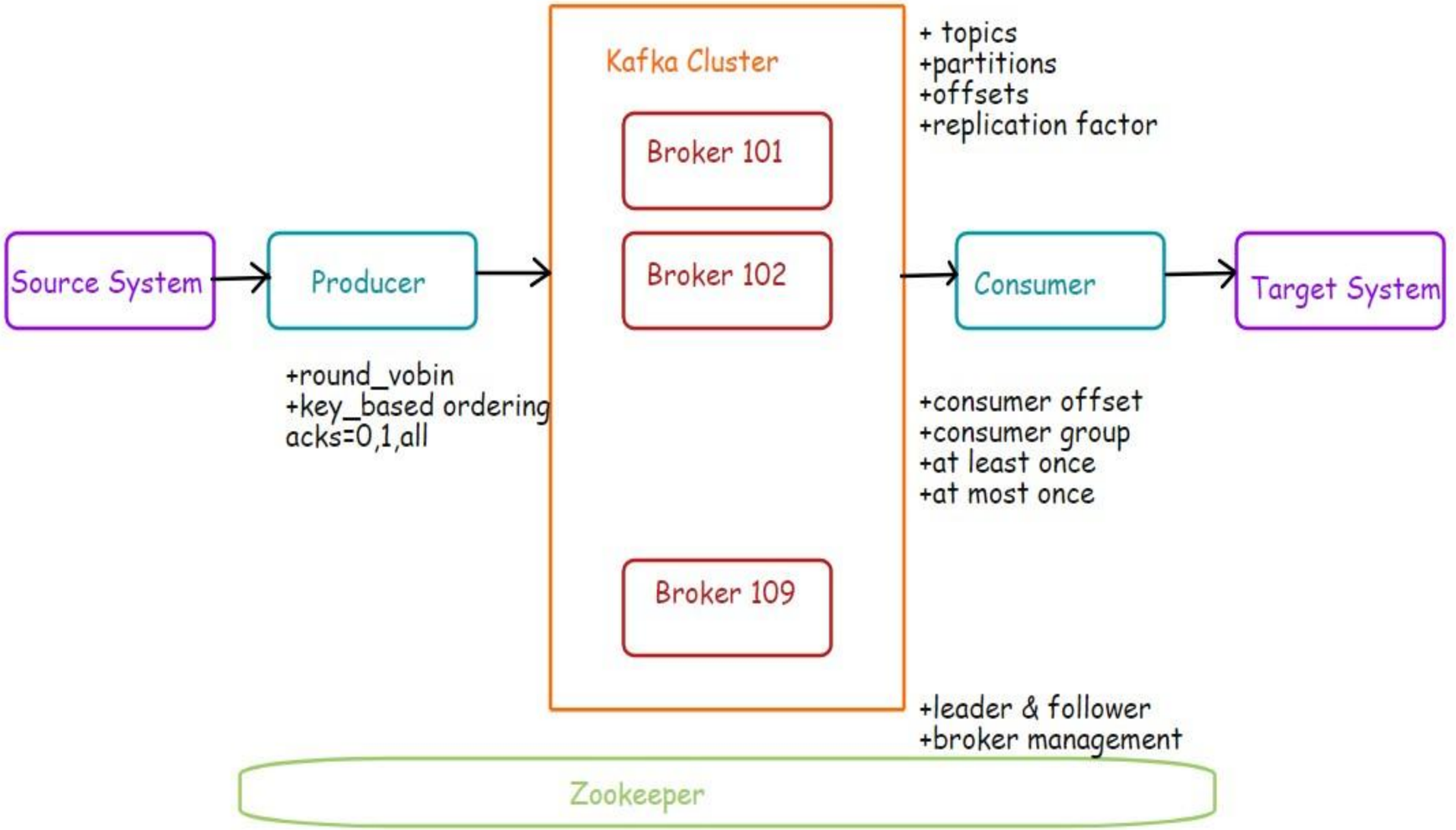
- Zookeeper manages the brokers (keeps a list of them)
- Zookeeper helps in performing leader elections for partitions
- Zookeeper sends notification to Kafka in case of
  - New topic
  - Broker dies
  - Broker comes up
  - Deleted Topic
  - Etc.
- Zookeeper operates with odd number of server

- Zookeeper has a leader (handle write) & rest of the serves are followers(handle read)



Kafka Guarantees

- Messages are appended to a topic-partition in order they are sent
- Consumer reads message in order stored in topic-partition
- With RF→N ; producers and consumers can tolerated N-1 brokers being going down
- RF-3 is a good idea
- As long as number of partitions remains constant; same key will go to same partition


Kafka Cluster
+ topics
+partitions
+offsets
+replication factor
Broker 101
Broker 102
Broker 109
Source System
Producer
Consumer
Target System
+round_vobin
+key_based ordering
acks=0,1,all
+consumer offset
+consumer group
+at least once
+at most once
+leader & follower
+broker management
Zookeeper

In the Logging Area, for Example, We Have Plugin For:

syslog

fluentd

Logstash

Filebeat

DBs

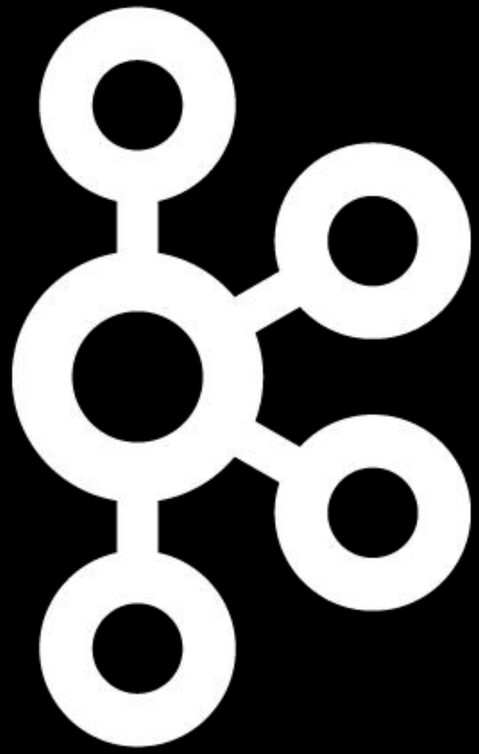
JDBC connector to cover RDBMS Dos

Debezium also offers a connector that include no-SQL Dos

Presto connector that allows us to query Kafka using Presto queries

We Learned

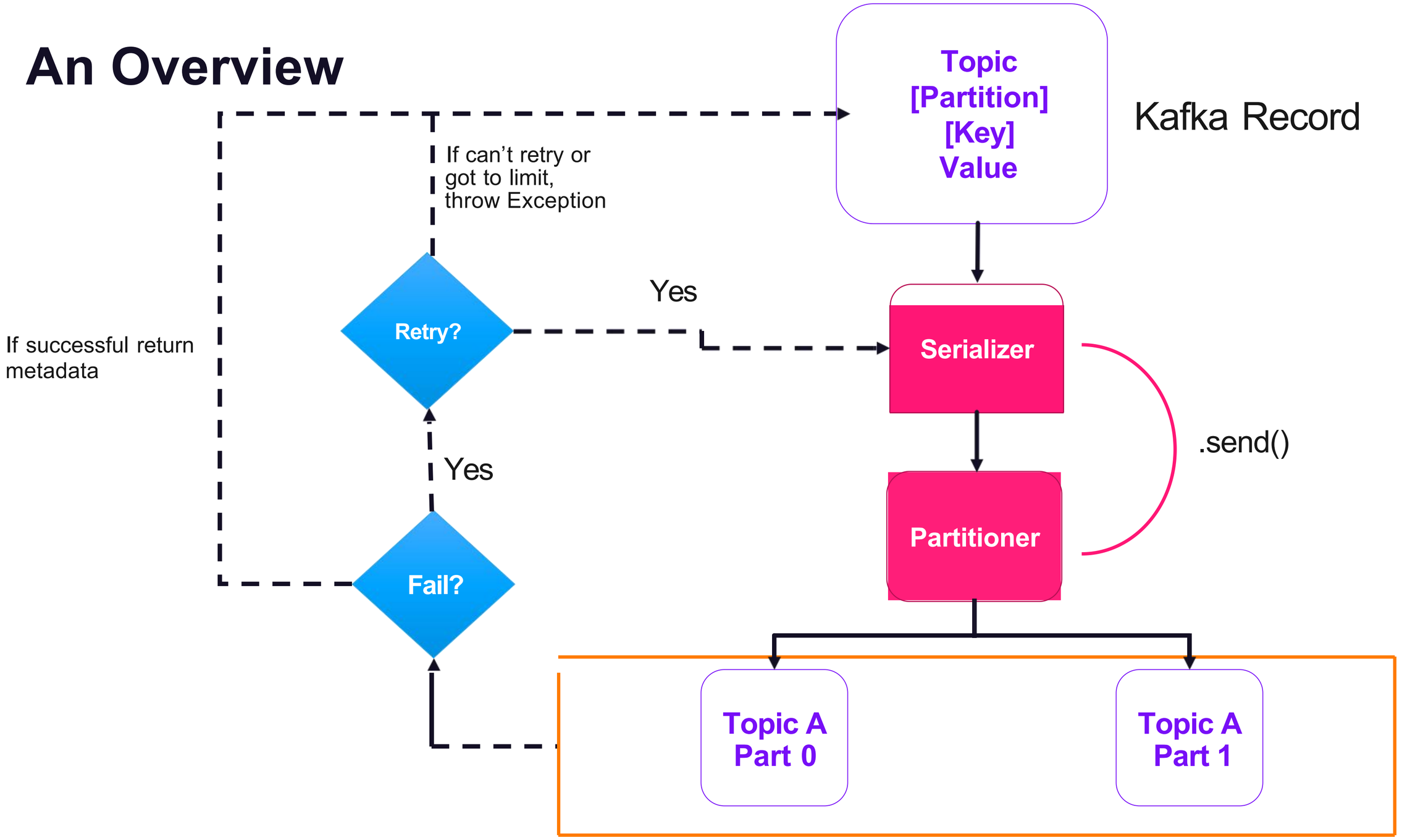
- Topics can be partitioned
- The partitioner uses the message key to redirect messages to partitions
- Consumers read from partitions and have an offset
- Therefore, more partitions enable more consumers, therefore more scalability

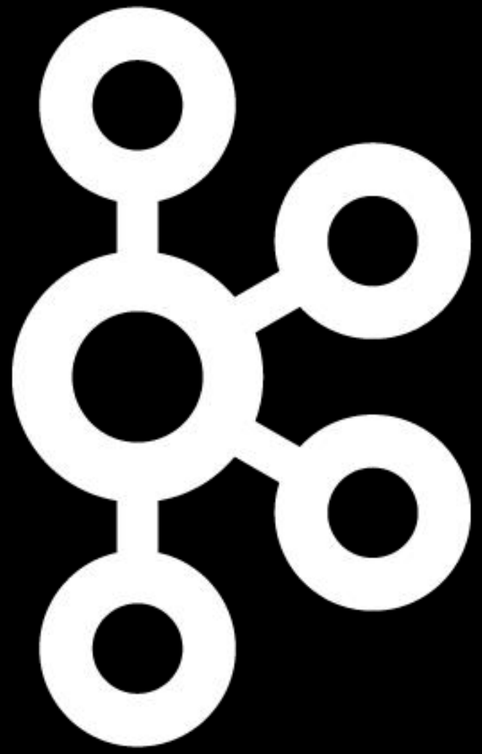


Deploy Kafka

Kafka Producers

An Overview


Topic
[Partition]
[Key]
Value
Kafka Record
If can't retry or
got to limit,
throw Exception
Retry?
Yes
If successful return
metadata
Serializer
.send()
Yes
Partitioner
Fail?
Topic A
Part 0
Topic A
Part 1



Producing Messages with Kafka CLI

Serializers and Producer Configuration

Construct a java.util Properties object

```
Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
```

Provide two or more locations where the Bootstrap servers are located

```
Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
```

Provide a Serializer for the key

```
Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
```

Provide a Serializer for the value

```
Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");

properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
```

Producer Object

```
KafkaProducer producer = new KafkaProducer<>(properties);
```

Create a Record

```
ProducerRecord producerRecord = new
ProducerRecord<>(“my_orders”, state, amount);
```

Sending a Message

```
Future send = producer.send(producerRecord);
```

Contains information about your send including the messages

Record Metadata

```
if (metadata.hasOffset()) {
      System.out.format("offset: %d\n",
      metadata.offset()),
}
      System.out.format("partition: %d\n",
      metadata.partition());

      System.out.format("timestamp: %d\n",
      metadata.timestamp());

      System.out.format("topic: %s\n",
metadata.topic());

      System.out.format("toString: %s\n",
metadata.toString());
```

Capturing a Callback

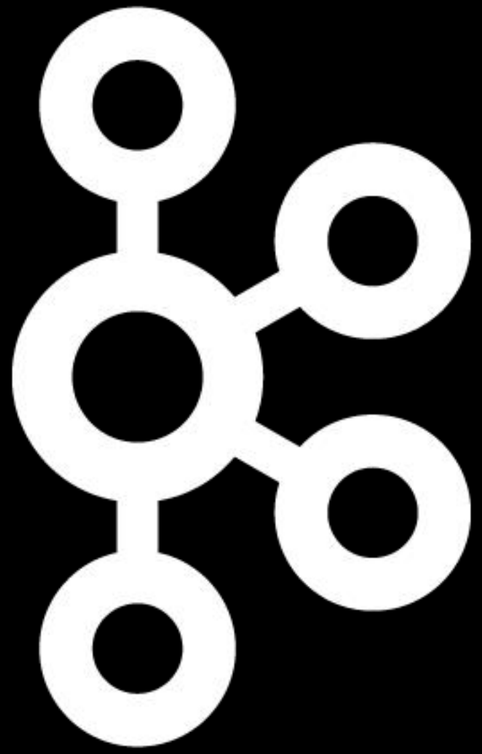
```
producer.send(producerRecord, new Callback() {

@Override

public void onCompletion(RecordMetadata metadata,
                         Exception e){


    }

}
```

Using Lambdas

```
producer.send(producerRecord,(metadata, e) -> {
    if(metadata != null) {

    System.out.println(producerRecord.key());
    System.out.println(producerRecord.value());

    }
}
```

Be a Good Citizen

```
producer.flush();
producer.close();
```



Processing Messages with Java

Key, Takeaways, and Tips

Takeaways

The producer protocol implies the existence of a Partitioner that redirects messages to the correct partition

The partitions information is caught on the initial instantiation of the Producer Object

The response from the producer.send() method is a Future, but you can capture it in a Callback or Lambda

There are Retryable and non-retryable Exceptions and based on that the Protocol will automatically retry

Keys

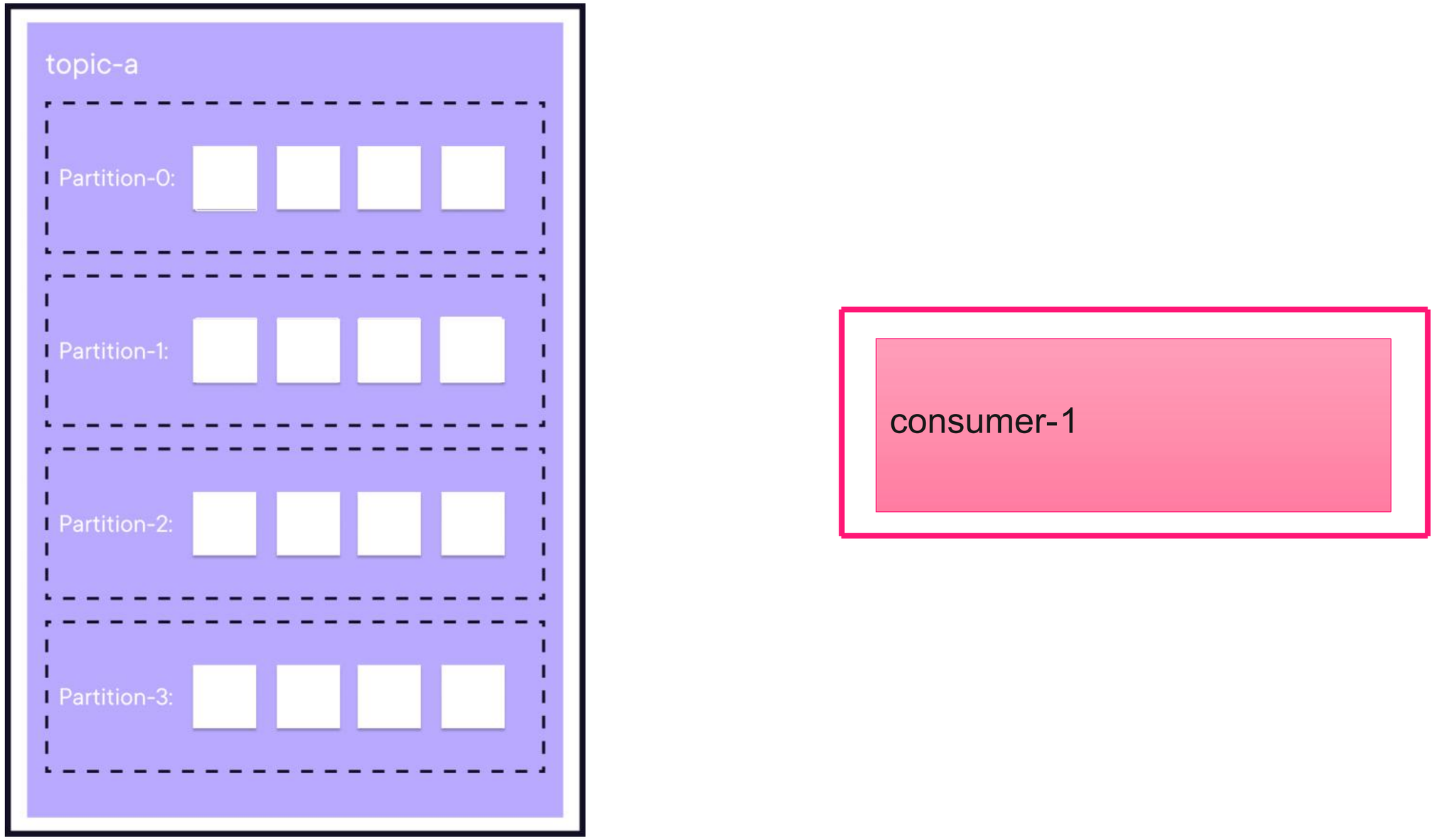
Be sure how to configure your own Partitioner as homework

Try to ensure you can write a simple loop of sending messages by yourself

Try playing around what happens with the Producer if a broker is dead

Kafka Consumers

Kafka Consumer Group


topic-a
Partition-0:
Partition-1:
Partition-2:
Partition-3:
consumer-1

Kafka Consumer Group

Consumers are tipically done as a group

A single consumer will end up inefficient with large amounts of data

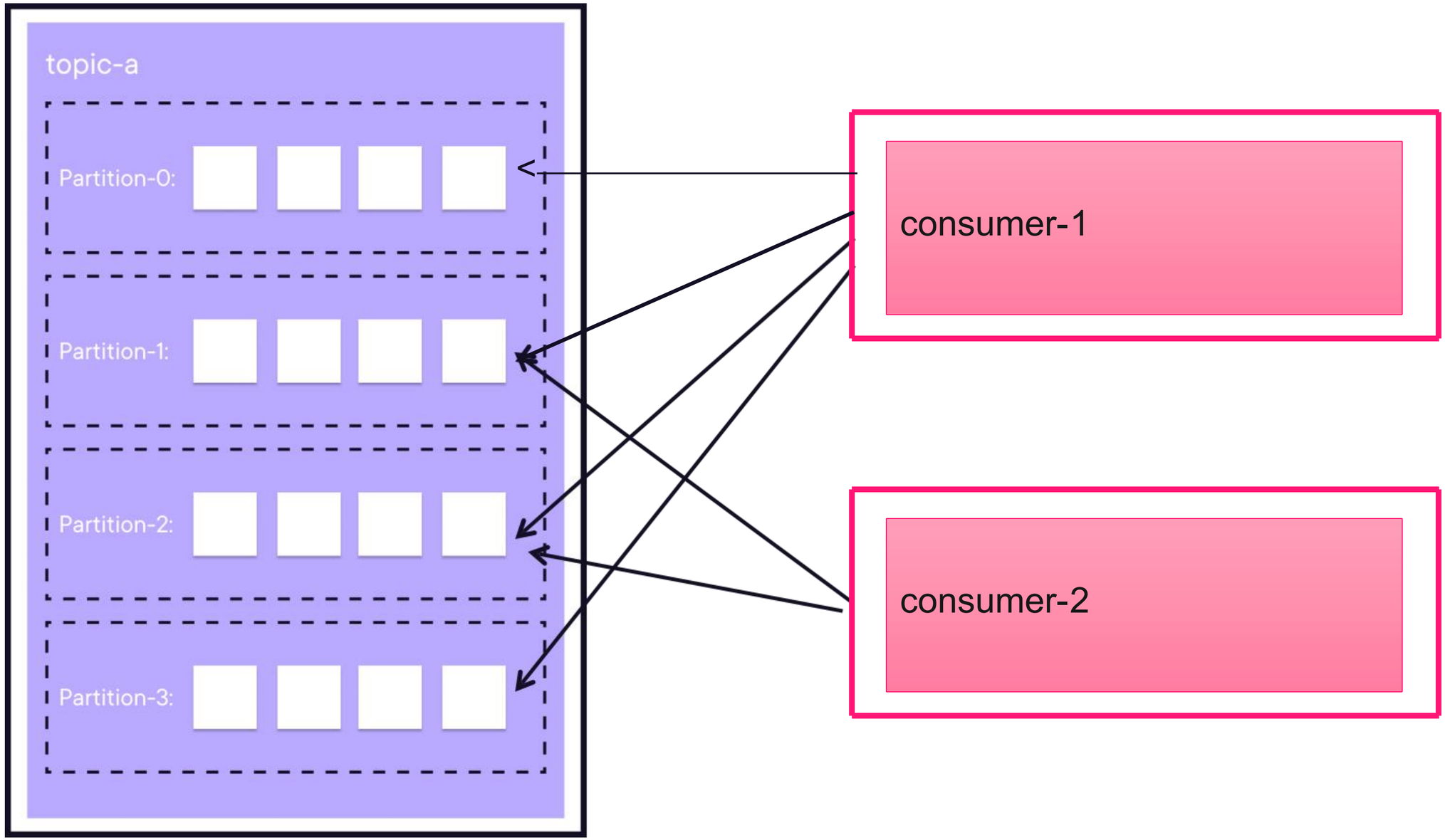
Kafka Consumer Group

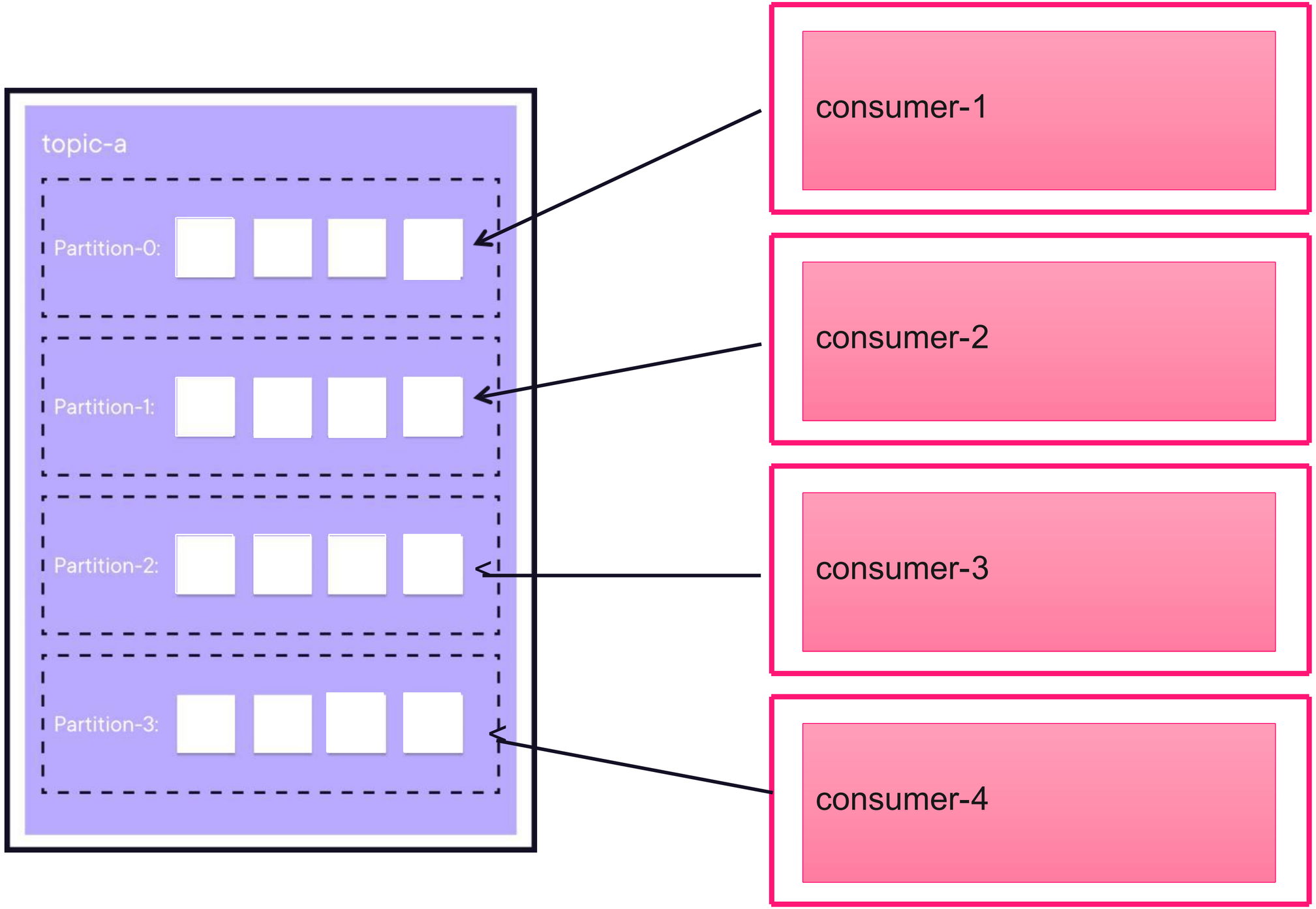
Consumers are tipically done as a group

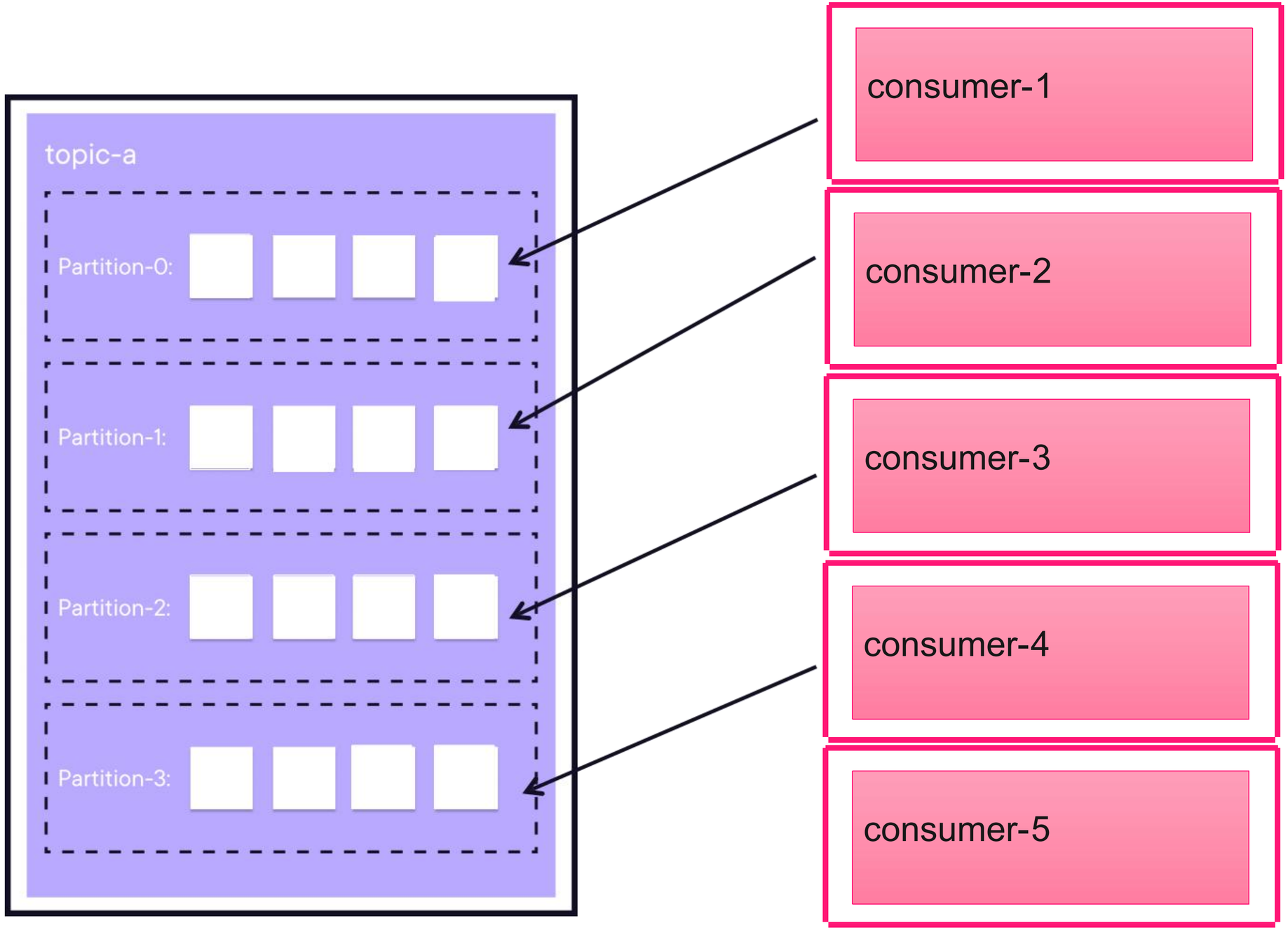
A single consumer will end up inefficient with large amounts of data

A consumer may never catch up

The consumer group id is key so Kafka knows that messages should be distributed to both consumers without duplicating


topic-a
Partition-0:
Partition-1:
Partition-2:
Partition-3:
consumer-1
consumer-2


topic-a
Partition-0:
Partition-1:
Partition-2:
Partition-3:
consumer-1
consumer-2
consumer-3
consumer-4


topic-a
Partition-0:
Partition-1:
Partition-2:
Partition-3:
consumer-1
consumer-2
consumer-3
consumer-4
consumer-5

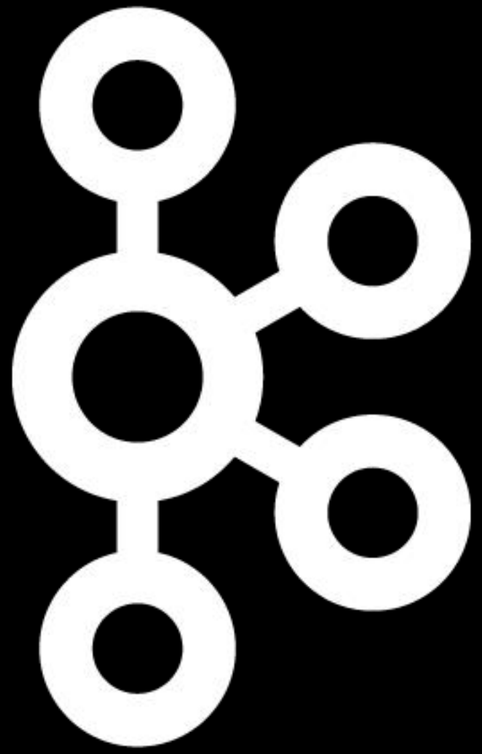
Kafka Consumer Group

Consumers are typically done as a group

A single consumer will end up inefficient with large amounts of data

A consumer may never catch up

Every consumer should be on it’s own machine, instance, pod



Consuming from Kafka

Deserializers and Consumer Configuration

Deserializers and Consumer Configuration

Construct a java.util.Properties object

```
Properties properties = new Properties();

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, “localhost:9092“);

properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringDeserializer");

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.IntegerDeserializer");

properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest“);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Deserializers and Consumer Configuration

Provide two or more locations where the Bootstrap servers are located

```
Properties properties = new Properties();

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, “localhost:9092“);

properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka
.common.serialization.StringDeserializer");

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kaf
ka.common.serialization.IntegerDeserializer");

properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest“);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Deserializers and Consumer Configuration

Provide a “Team Name”, officially called a group.id

```
Properties properties = new Properties();

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, “localhost:9092“);

properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka
.common.serialization.StringDeserializer");

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kaf
ka.common.serialization.IntegerDeserializer");

properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest“);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Deserializers and Consumer Configuration

Provide a DeSerializer for the key

```
Properties properties = new Properties();

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, “localhost:9092“);

properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka
.common.serialization.StringDeserializer");

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kaf
ka.common.serialization.IntegerDeserializer");

properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest“);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Deserializers and Consumer Configuration

Provide a DeSerializer for the value

```
Properties properties = new Properties();

properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, “localhost:9092“);

properties.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group");

properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,"org.apache.kafka
.common.serialization.StringDeserializer");

properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,"org.apache.kaf
ka.common.serialization.IntegerDeserializer");

properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest“);

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
```

Consumer Object

Construct a org.apache.kafka.clients.consumer.KafkaConsumer object

```
KafkaConsumer consumer = new KafkaConsumer<>(properties);
```

Processing

Use the consumer that you have constructed, and call poll with pulse time.

```
while (!done.get()) {
        ConsumerRecords<String, String> records =
        consumer.poll(Duration.of(500, ChronoUnit.MILLIS));
    for (ConsumerRecord<String, String> record : records) {
        System.out.format("offset: %d\n", record.offset());
        System.out.format("partition: %d\n", record.partition());
        System.out.format("timestamp: %d\n", record.timestamp());
        System.out.format("timeStampType: %s\n",
record.timestampType());
         System.out.format("topic: %s\n", record.topic());
         System.out.format("key: %s\n", record.key());
         System.out.format("value: %s\n", record.value());
    }
}
```

Be a Good Citizen

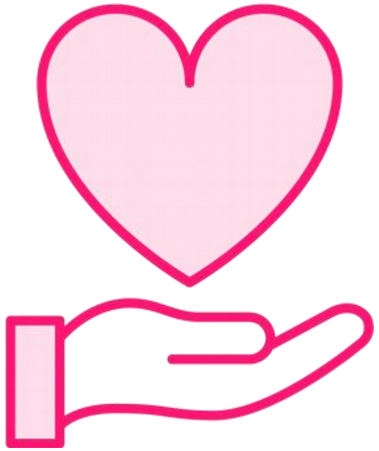
Processing

```
consumer.close();
```

Speaking of Rebalances

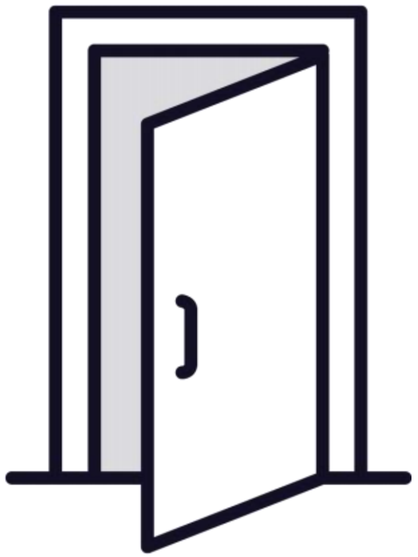
Regardless of what a consumer is doing, at regular intervals, which are configurable, they send a heartbeat to Kafka

Specifically to the coordinator inside the broker, basically letting them know they are alive

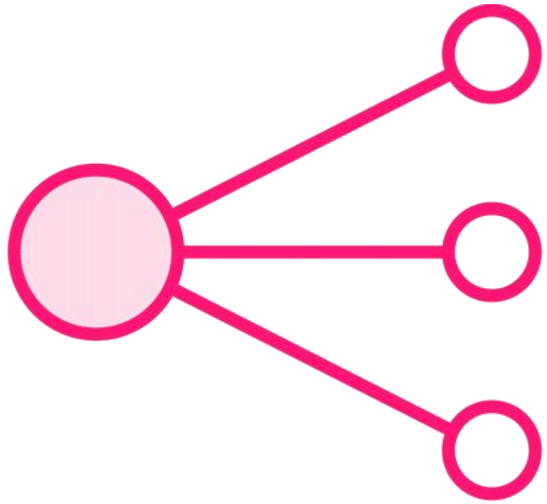


Also when we poll for records that lets Kafka know that consumer is alive

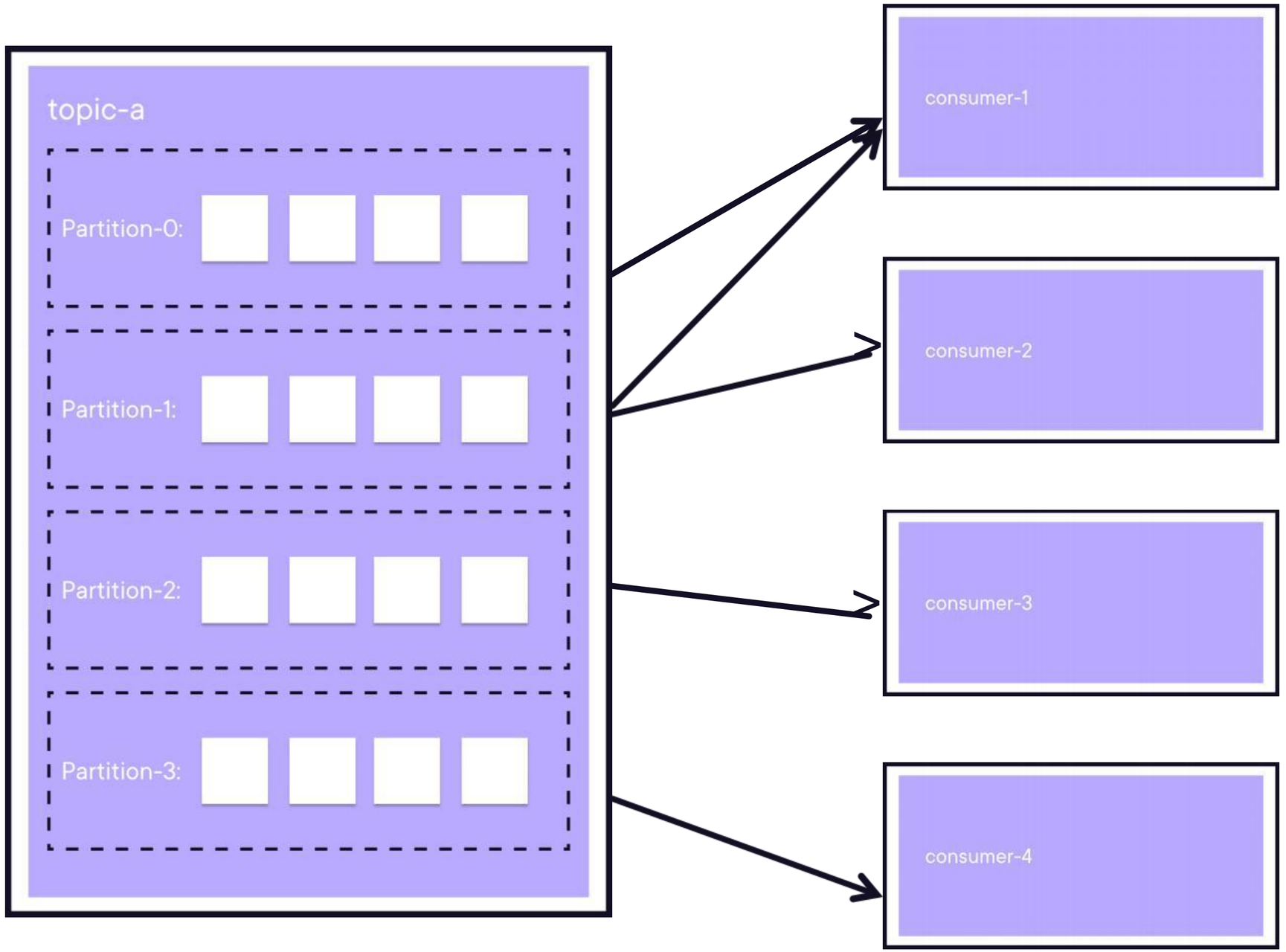
Speaking of Rebalances

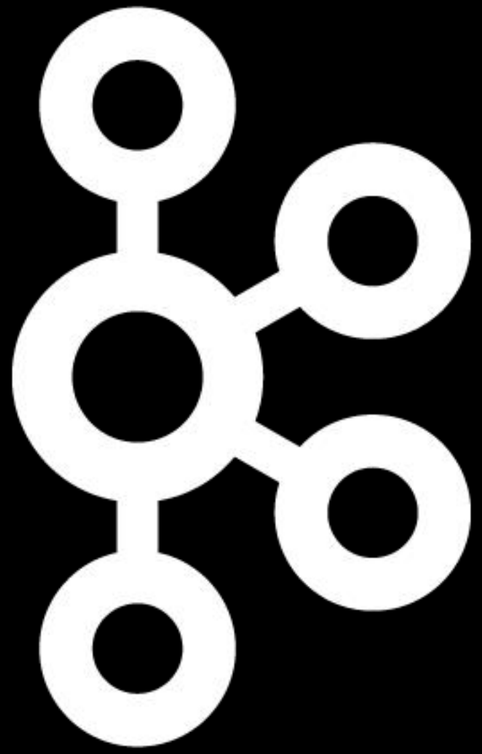


And as we said when you close a consumer that close method lets Kafka know they are not alive anymore



Which triggers a rebalance


topic-a
Partition-0:
Partition-1:
Partition-2:
Partition-3:
consumer-1
consumer-2
consumer-3
consumer-4



Consuming Messages with Java

Key, Takeaways, and Tips

Takeaways

Consumers act as a group via the Consumer group ID

Each consumer in the group gets assigned a partition, and a partition is not shared by two members of a group

A rebalance is triggered when a member leaves the group or they haven’t sent a heartbeat in a long time

A rebalance is a stop the world event that ensures all partitions are attended by some consumer in the group

Keys

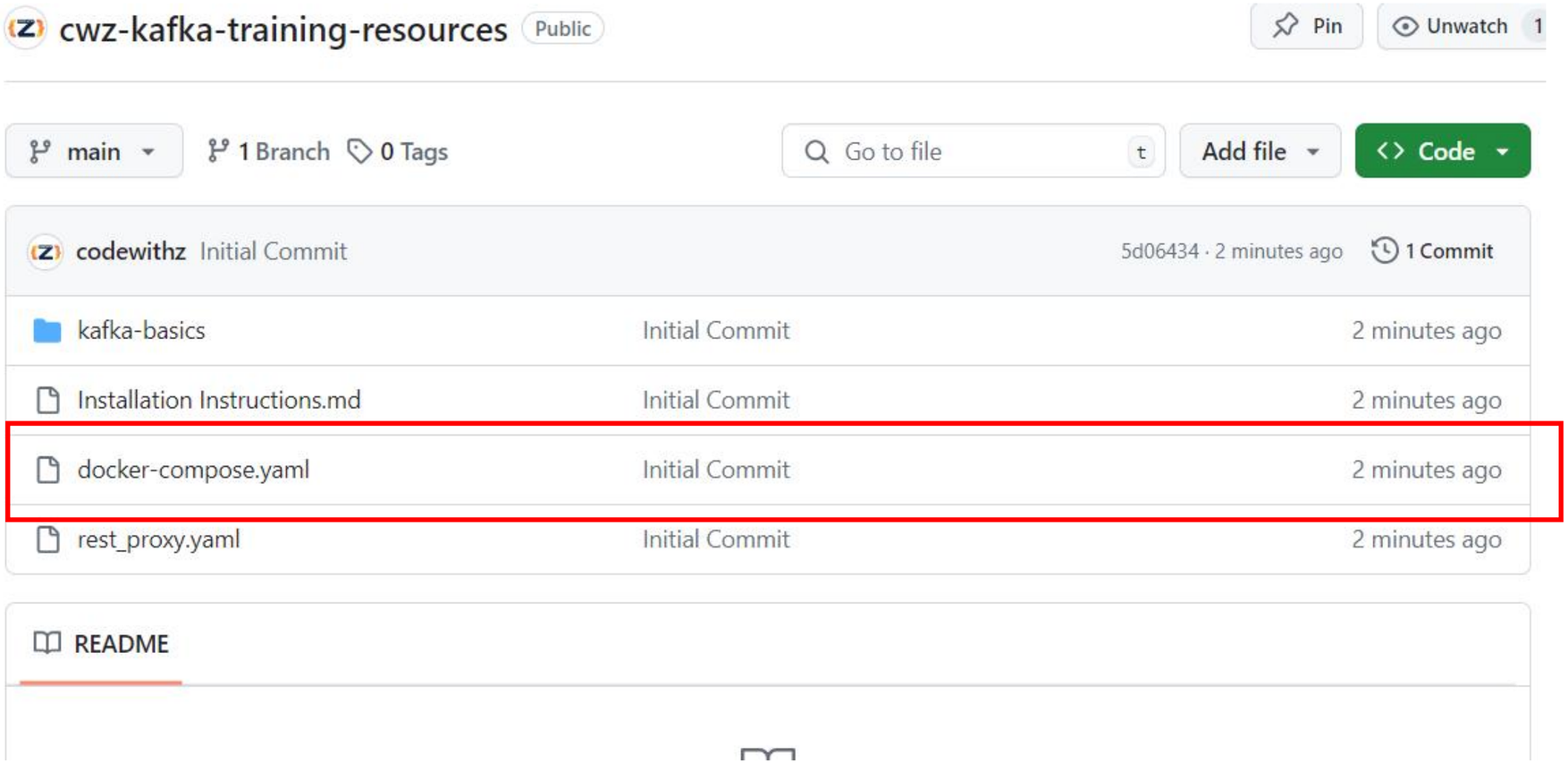
Be sure to configure your Consumer to ensure you are not duplicating messages

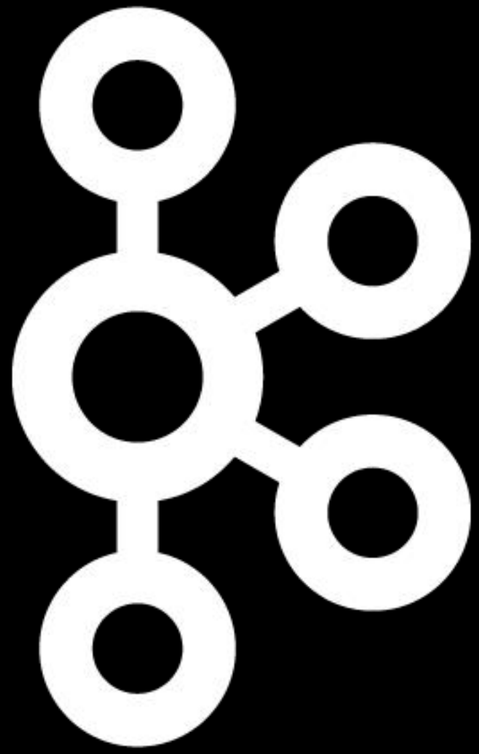
Try to create a consumer loop with two consumers and verify the partition assignament playing with keys in the producer

See what happens in the logs of the broker when a consumer joins or leaves the group

Launch Kafka Using the docker-compose file on following link

https://github.com/codewithz/cwz-kafka-training-resources





Kafka CLI

If using Linux, add .sh after kafka-topics

```
#-------------------------------------------
#---------- To Create a Topic [Zookeper]
#-------------------------------------------

kafka-topics --zookeeper host:port --topic name_of_topic --create
--partitions noOfPartitions --replication-factor noOfReplicas

#-------------------------------------------
#---------- To Create a Topic [Bootstrap Server]
#-------------------------------------------

kafka-topics --bootstrap-server host:port --topic name_of_topic --create
--partitions noOfPartitions --replication-factor noOfReplicas

kafka-topics --bootstrap-server localhost:9092 --topic bot_first_topic --create --partitions 3
--replication-factor 1

#
```

```
#-------------------------------------------
#---------- To List the Topics [Bootstrap Server]
#-------------------------------------------

kafka-topics --bootstrap-server host:port --List

kafka-topics -bootstrap-server localhost:9092 --list

#-------------------------------------------
#---------- To Describe a Topic [Bootstrap Server]
#-------------------------------------------

kafka-topics --bootstrap-server host:port --topic name_of_topic --describe

kafka-topics --bootstrap-server localhost:9092 --topic bot_first_topic --describe
```

```
#-------------------------------------------
#---------- Delete a Topic [Bootstrap Server]
#-------------------------------------------

kafka-topics --bootstrap-server host:port --topic name_of_topic --delete

kafka-topics --bootstrap-server localhost:9092 --topic bot_first_topic --delete
```

```
#-------------------------------------------
#---------- To Produce Data to a Topic [Bootstrap Server]
#-------------------------------------------

kafka-console-producer --bootstrap-server host:port
--topic name_of_topic

kafka-console-producer --bootstrap-server localhost:9092
--topic bot_first_topic
```

```
#-------------------------------------
#---------- To Produce Data to a Topic with properties
#-------------------------------------

kafka-console-producer --bootstrap-server host:port
--topic name_of_topic --producer-property key=value

kafka-console-producer --bootstrap-server localhost:9092
--topic bot_first_topic --producer-property acks=all
```

```
#----------------------------------------
#---------- To Produce Data to a Non-Existing Topic [Bootstrap Server]
#----------------------------------------

kafka-console-producer --bootstrap-server host:port
--topic name_of_topic_which_doesnt_exists

kafka-console-producer --bootstrap-server localhost:9092
--topic bot_fourth_topic

#### ---------- ALWAYS CREATE A TOPIC BEFORE PRODUCING DATA TO THEM --
```

```
#------------------------------------
#---------- To Consume Data from a Topic
#------------------------------------

kafka-console-consumer --bootstrap-server host:port
--topic name_of_topic

kafka-console-consumer --bootstrap-server localhost:9092
--topic bot_first_topic
```

```
#------------------------------------
#---------- To Consume Data from a Topic from beginning
#------------------------------------

kafka-console-consumer --bootstrap-server host:port
--topic name_of_topic --from-beginning

kafka-console-consumer --bootstrap-server localhost:9092
--topic bot_first_topic --from-beginning
```

```
kafka-log-dirs.sh --bootstrap-
server <kafka-broker> --describe --
broker-list <broker-id>
```

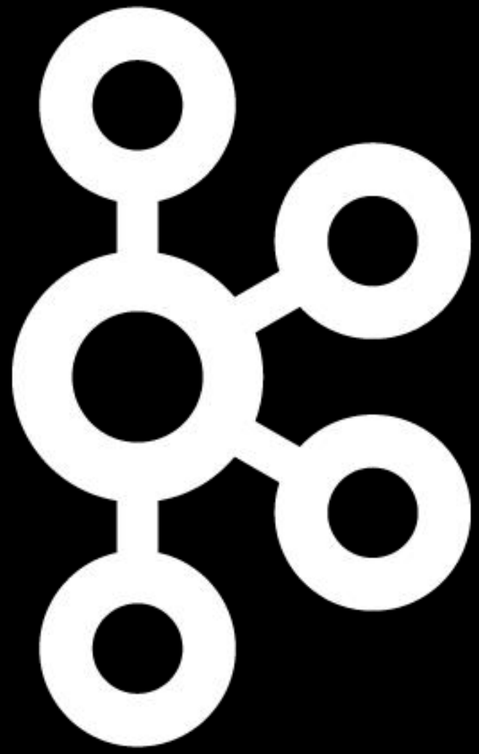
```
#-------------------------------------------
#---------- To Consume Data from a Topic
#-------------------------------------------

kafka-console-consumer --bootstrap-server host:port
--topic name_of_topic --group groupName


kafka-console-consumer --bootstrap-server localhost:9092
--topic bot_second_topic --group my-first-app

kafka-console-consumer --bootstrap-server localhost:9092
--topic bot_second_topic --group my-first-app
```

```
kafka-consumer-groups --bootstrap-server localhost:9092 --list
```

```
kafka-consumer-groups --bootstrap-server localhost:9092
--describe --group my-first-app
```

```
GROUP           TOPIC                PARTITION  CURRENT-OFFSET  LOG-END-
my-first-app    bot_second_topic 0           8               8
my-first-app    bot_second_topic 1           8               8
my-first-app    bot_second_topic 2           8               8
```

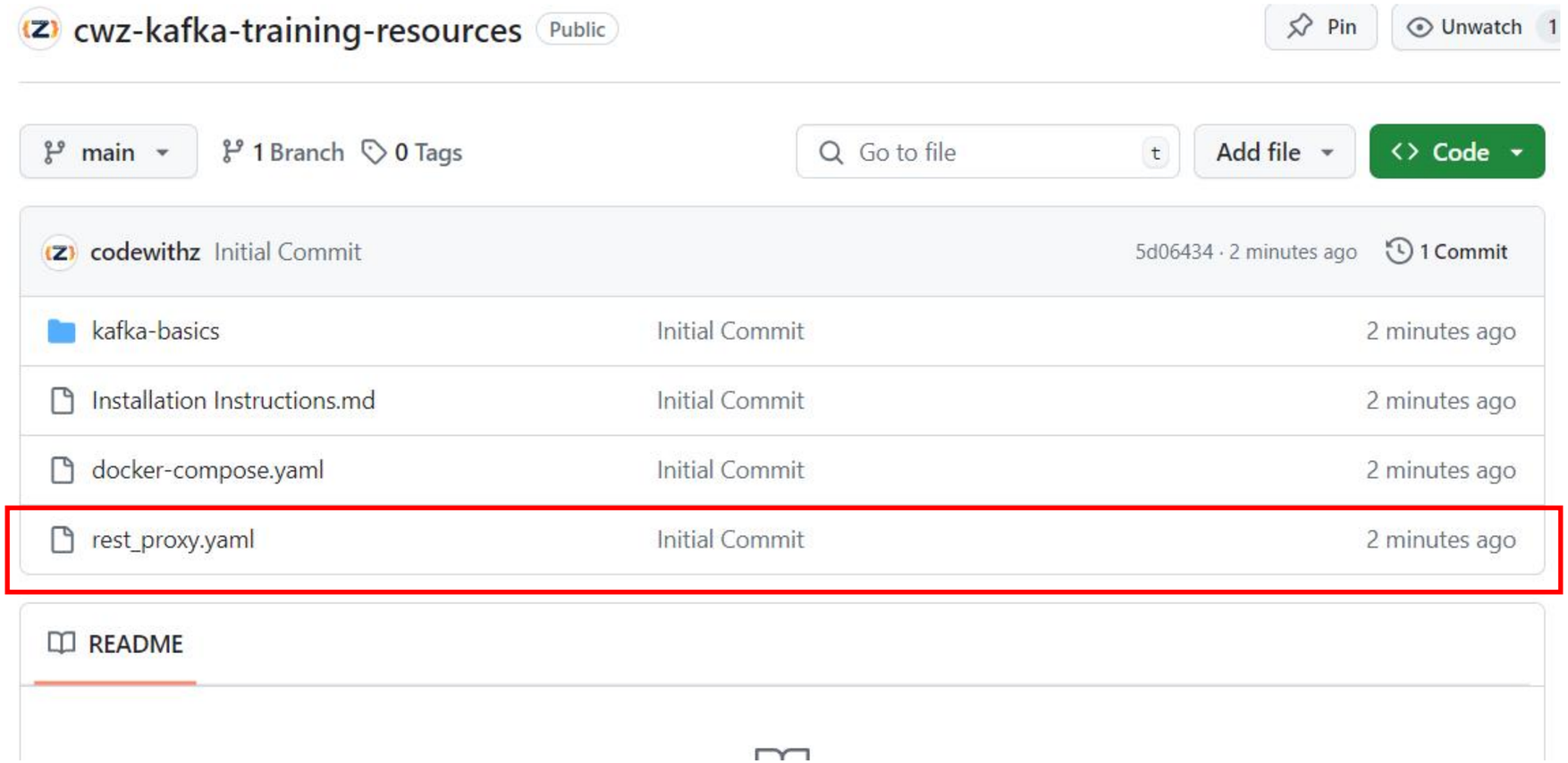
```
#-----------------------------------------
#---------- Resetting Offset
#-----------------------------------------

kafka-consumer-groups --bootstrap-server localhost:9092
--group my-first-app --reset-offsets
--to-earliest --execute --topic bot_first_topic
```

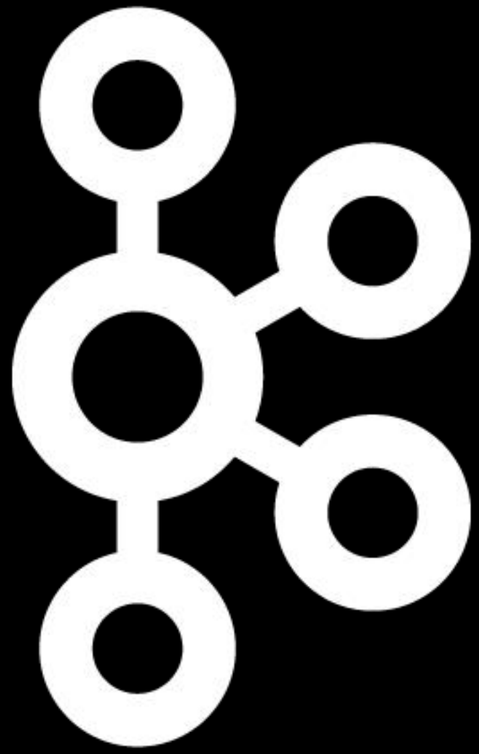


Kafka Rest Proxy

Import rest_proxy.yaml from the git repo and import to POSTMAN

https://github.com/codewithz/cwz-kafka-training-resources

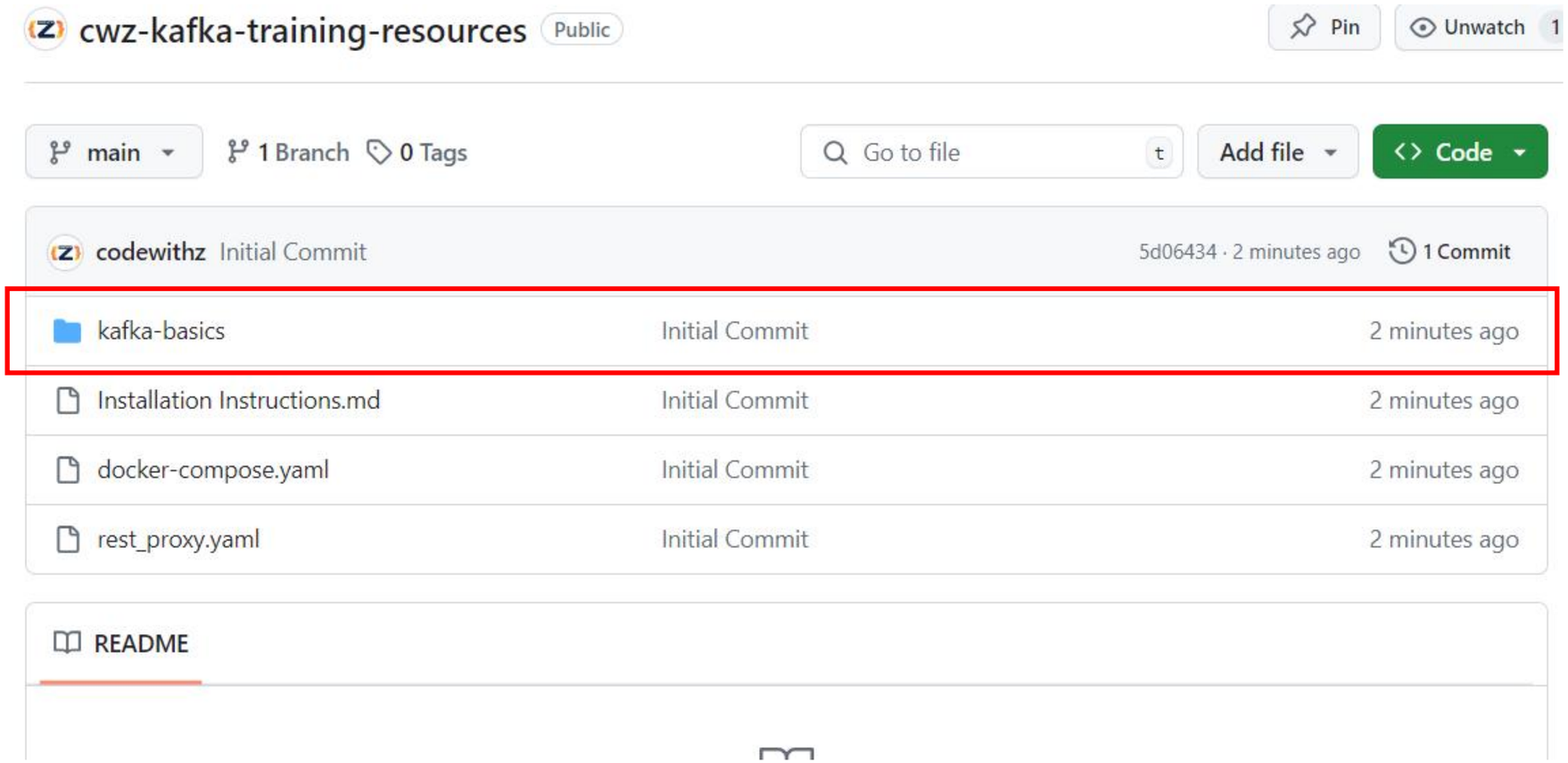




Kafka Java Producer and Consumer

Download the Skeleton Project from following link.

https://github.com/codewithz/cwz-kafka-training-resources

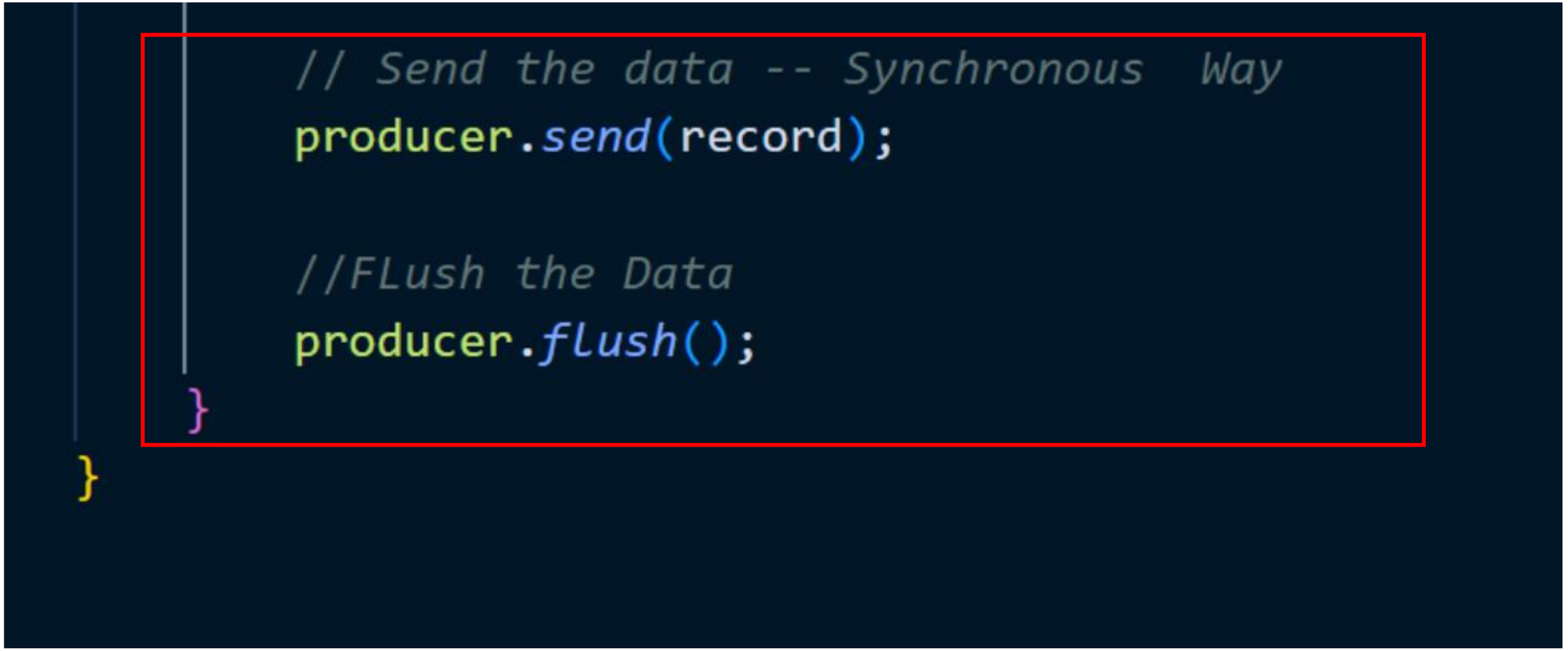


Create a Java File by name ProducerDemo.java

```java
package com.codewithz;


import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import java.util.Properties;

public class ProducerDemo {

    Run | Debug
    public static void main(String[] args) {
        String bootstrapServer="127.0.0.1:9092";
        String topic="codewithz_first_topic";
        //Create Producer Properties
```

```
//Create Producer Properties

Properties properties=new Properties();
properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,bootstrapServer);
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSerializer.class.getName());
```

```
// Create the Producer

KafkaProducer<String,String> producer=new KafkaProducer<String, String>(properties);

//Create a ProducerRecord
ProducerRecord<String,String> record=new ProducerRecord<>(topic,value:"Hello from Java Code");
```


// Send the data -- Synchronous Way
producer.send(record);
//FLush the Data
producer.flush();
}
}

Create a Java File by name ProducerDemoWithCallback.java

```java
package com.codewithz;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;

public class ProducerDemoWithCallback {

    Run | Debug
    public static void main(String[] args) {
        String bootstrapServer="127.0.0.1:9092";

        Logger logger= LoggerFactory.getLogger(ProducerDemoWithCallback.class);
        String topic ="codewithz_third_topic";
```

```
//Create Producer Properties

Properties properties=new Properties();
properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServer);
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

```java
// Create the Producer

KafkaProducer<String,String> producer=new KafkaProducer<String, String>(properties);

for (int i=1;i<=10;i++){
    //Create a Producer Record
    ProducerRecord<String,String> record=
    new ProducerRecord<>(topic,"Hello from Java Code: "+Integer.toString(i));
```

Sending Data in Asynchronous Way

```
producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata recordMetadata, Exception e) {
                // Execute everytime a record is successfully sent,
                //if record sending fails, it will throw an exception
                if(e==null){
                    //Record is succesfully sent
                    logger.info("Recieved new MetaData");
                    logger.info("Topic:"+recordMetadata.topic()+"\n"+
                            "Partition:"+recordMetadata.partition()+"\n"+
                            "Offset:"+recordMetadata.offset()+"\n"+
                            "Timestamp:"+recordMetadata.timestamp()
                    );
                }
                else{
                    //Exception have occurred
                    logger.error("Error while producing :",e);
                }
            }
        }
);
```

```
            );
        }

            //Flush the Data
            producer.flush();

            //Flush and CLose Producer
            producer.close();

    }
}
```

Create a Java File by name ProducerDemoWithKeys.java

```java
package com.codewithz;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;


public class ProducerDemoWithKeys {

    Run | Debug
    public static void main(String[] args) {
        String bootstrapServer="127.0.0.1:9092";


        Logger logger= LoggerFactory.getLogger(ProducerDemoWithKeys.class);
        String topic ="codewithz_third_topic";
```

```
//Create Producer Properties

Properties properties=new Properties();
properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServer);
properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

```java
KafkaProducer<String,String> producer=new KafkaProducer<String, String>(properties);
```

```java
for (int i=1;i<=10;i++){
    //key and Value
    String key="id_"+Integer.toString(i);
    String value="Hello from Java Code: "+Integer.toString(i);
    //Create a Producer Record
    ProducerRecord<String,String> record=new ProducerRecord<>(topic,key,value);

    logger.info("Key:"+key);
    //id_1 -- P0
    //id_2 -- P2
    //id_3 -- P0
    //id_4 -- P2
    //id_5 -- P2
    //id_6 -- P0
    //id_7 -- P2
    //id_8 -- P1
```

```java
// Send the data -- Asynchronous way

producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata recordMetadata, Exception e) {
                // Execute everytime a record is successfully sent,
                //  if record sending fails, it will throw an exception
```

```
if(e==null){
    //Record is succesfully sent
    logger.info("Recieved new MetaData");
    logger.info("-----------------------------------"+"\n"+
            "Topic:"+recordMetadata.topic()+"\n"+
            "Partition:"+recordMetadata.partition()+"\n"+
            "Offset:"+recordMetadata.offset()+"\n"+
            "Timestamp:"+recordMetadata.timestamp()+"\n"+
            "Key:"+key+"\n"+
                "-----------------------------------"


    );
}
```

```
                    else{
                        //Exception have occurred
                        logger.error("Error while producing :",e);
                    }
                }
            }
        );
    }

    //Flush the Data
    producer.flush();

    //Flush and CLose Producer
    producer.close();
```

Create a Java File by name ConsumerDemo.java

```java
package com.codewithz;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.Arrays;
import java.util.Collections;
import java.util.Properties;

public class ConsumerDemo {

    Run | Debug
    public static void main(String[] args) {

        Logger logger= LoggerFactory.getLogger(ConsumerDemo.class);
        String bootstrapServer="localhost:9092";
        String groupId="my-third-app";
```

```
//Create Consumer Configs
Properties properties=new Properties();
properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServer);
properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
 StringDeserializer.class.getName());
properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG,groupId);
properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
value:"earliest"); // earliest/latest/none
```

```
//Create a Consumer
KafkaConsumer<String,String> consumer=
new KafkaConsumer<String, String>(properties);
```

```
    //Subscribe to the topic

    String topic="codewithz_third_topic";

/       consumer.subscribe(Collections.singleton(topic));

    consumer.subscribe(Arrays.asList(topic));
```

```
        //Poll for New Data

        while (true){
            ConsumerRecords<String,String> records=
            consumer.poll(Duration.ofMillis(millis:100));

            for(ConsumerRecord<String,String> record:records){
                logger.info("-----------------------------------------");
                logger.info("Key:"+record.key()+" | Value:"+record.value());
                logger.info("Partition:"+record.partition()+" | Offset:"+record.offset());
            }
        }
    }
}
```