# MX2020 Big Data

## Event Hub Reference Architecture

Mayo 10, 2019

# Change control

| Versión | Fecha | Descripción del Cambio | Autor/Departamento |
|---|---|---|---|
| 0.1 | 10/05/2019 | Creation of the document | [Big Data Architecture] |
| | | | |
| | | | |
| | | | |
| | | | |

# Table of Contents

# Needs and Solution Proposal (I)

## Descripción de la solución técnica

### Objective

- Define core components, pieces of code for the CitiBanamex Even Hub Reference Architecture

### Scope

- Graphically represent the components, their integrations and the use of good practices in the development of software engineering processes.

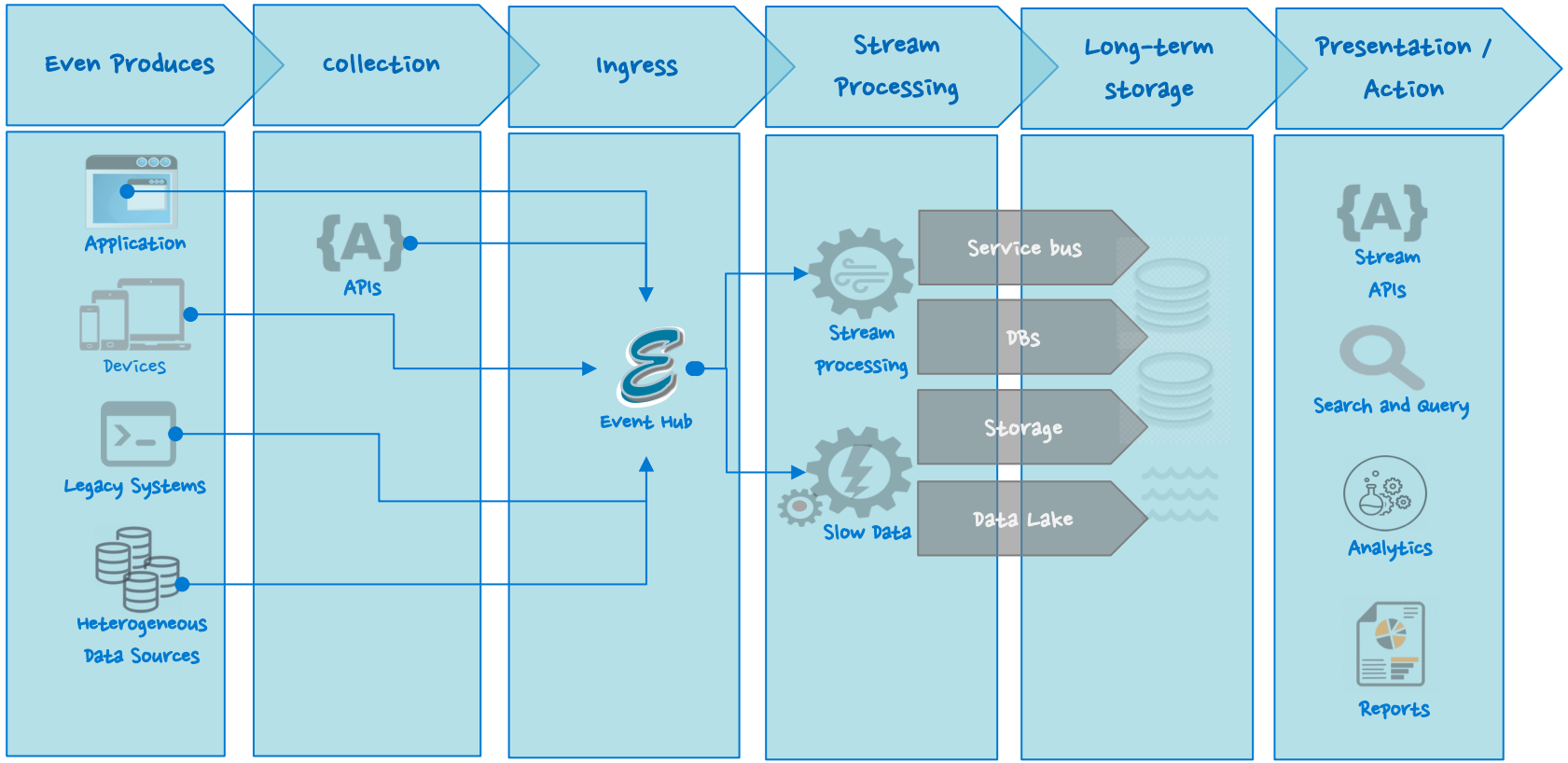
### Description

- Event Hub is a highly scalable data streaming platform and event ingestion service, capable of receiving and procession millions of events per second. Event Hub can process and store events, data or telemetry produced by distributed software and devices. Data sent to an event hub can be transformed and stored using any real-time analytics provider or batching/storage adapters. With the ability to provide publish-subscribe capabilities with low latency and at massive scale, Event Hubs services as the “on-ramp” for Big Data.
- Even Hub and telemetry handling capabilities make it especially useful for:
  - ✓ Application instrumentation
  - ✓ User expediencies or workflow processing
  - ✓ Enable behavior tracking in mobile apps, traffic information from web farms, connected with other devices

# Needs and Solution Proposal (I)

## Event Hub Architecture

Event Hub channels the events. So Event Hub is a component or services that is located between event editors and event consumers to decouple the production of event flow from the consumption of those events. The following figure represents this architecture:




Event Hub feature following key elements:

**Event producers/publishers:** An entity that sends data to an event hub.

**capture:** Enables you to capture Event Hubs streaming data and store it in an Azure Blob storage account

**Partitions:** Enables each consumer to only read a specific subset, or partition, of the event stream.

**Tokens:** Identifies and authenticates the event publisher

**Event consumers:** An entity that reads event data from an event hub

**consumer groups :** Provides each multiple consuming application with a separate view of the event stream, enabling those consumers to act independently.

**Throughput units:** Pre-purchased units of capacity. A single partition has a maximum scale of 1 throughput unit.

# Needs and Solution Proposal (II)

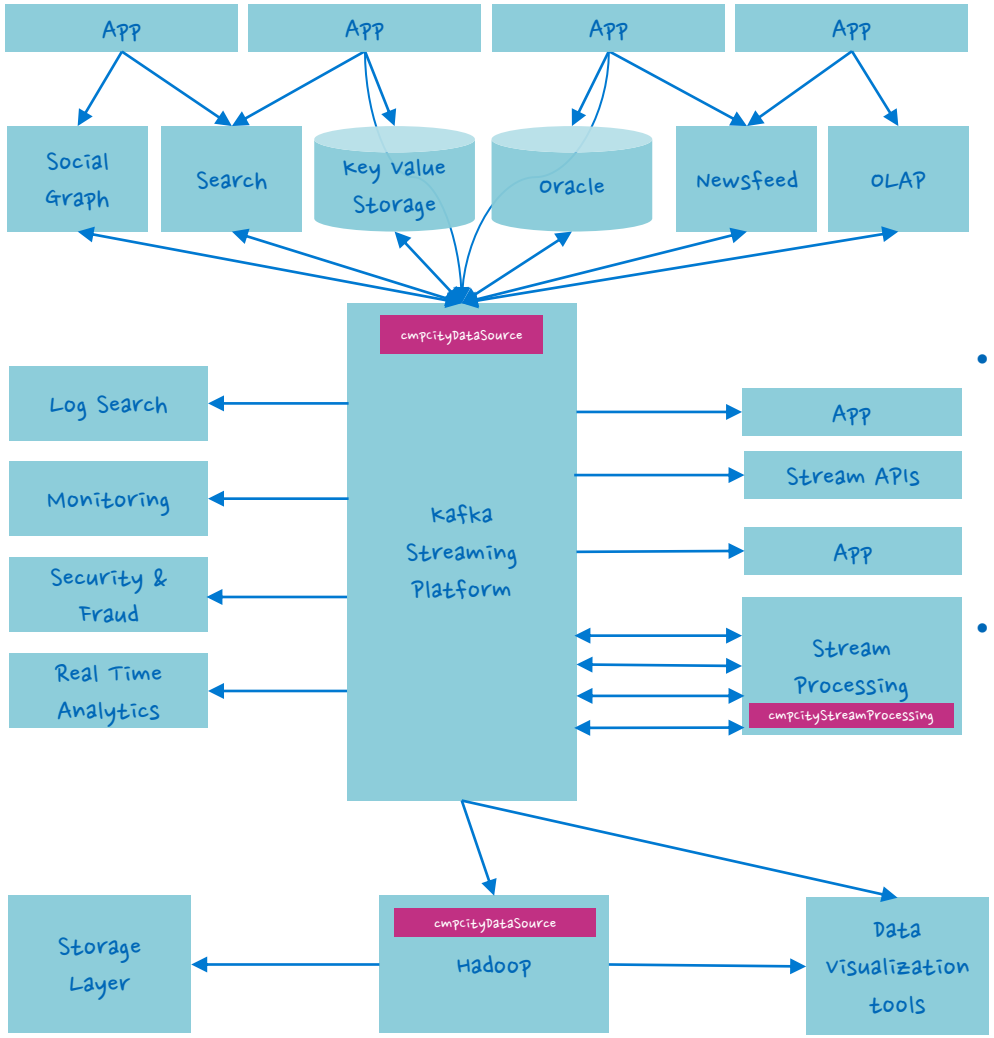
## Event Hub Architecture – Event streaming platform

### A modern stream-centric data architecture

In Event-Hub kafka acts as a type of universal channeling for data. Each system can feed this central duct or be fed by it; the applications or flow processors can access it to create new derived flows, which in turn can be fed back into the various systems to serve.

### Principles of Even Hub

- Most of what citiBanamex does can be considered as a series of events
- Data is more than rows in databases, they are events




A modern stream-centric data architecture

### An event streaming platform has two primary uses:

- Stream processing: It enables continuous, real-time applications built to react to, process, or transform streams. This is the natural evolution of the world of Enterprise Messaging which focused on single message delivery, stream processing gives you that and more.
- Data Integration: The event streaming platform captures streams of events or data changes and feeds these to other data systems such as relational databases, key-value stores, Hadoop, or

# Needs and Solution Proposal (III)

## Event Hub Architecture – The key capabilities of an Event Hub

| Event Hub | Description |
|---|---|
| Pub / Sub | Enable real-time publishing and subscribing to data streams at scale, in particular:<br>• Provide single-digit latency so it is usable by real-time applications<br>• Able to handle high volume data streams such as are common for log data, or other usage across many applications<br>• Enable integration with passive data systems such as databases as well as applications that actively publish events |
| Process | Enable the real-time stream processing of streams at scale<br>Support modern stream processing capabilities to power real-time applications or streaming transformations |
| Store | Be able to reliably store streams of data at scale. In particular:<br>It provides solid guarantees of replication and order to handle critical updates, such as replicating the record of changes in a database in a replica store as a search index, delivering this data in order and without loss.<br>It is capable of storing or storing data for long periods of time. This allows integration with batch systems (such as a data warehouse) that only load data periodically, as well as allowing the flow processing systems to reprocess the sources when their logic changes.<br>Storage must be economical enough for the platform to be viable at scale. |

# Needs and Solution Proposal (IV)

## Event Hub Architecture – The key capabilities of an Event Hub

Points to highlight about the relationship this event streaming platform concept has with other.

| System | Description |
|---|---|
| Messaging | Even Hub is similar to an enterprise messaging system: it receives messages and distributes to interested subscribers. It is considered as a type of messaging 2.0. There are three important differences:<br>• Messaging systems usually run in single implementations for different applications. Because Even Hub runs as a cluster and can be scaled, a central instance can support a complete data center<br>• The messaging systems have limitations in the work of data storage and this limits them to flows that only have consumption in real time. However, many integrations require more than this<br>• The semantics of messaging systems are not easily compatible with rich flow processing, so they really can not be used as the basis for the processing of Even Hub |
| Data Integration Tools | Enable the real-time stream processing of streams at scale<br>Support modern stream processing capabilities to power real-time applications or streaming transformations<br>One practical area of overlap is that by making the data available in a uniform format in one place with a common abstraction stream, many of the routine data cleaning tasks can be completely avoided |
| Enterprise Service Buses | The advantage of Even Hub is that the transformation is fundamentally decoupled from the flow itself. This code can live in applications or transmit processing tasks, allowing teams to go their own pace without a central bottleneck for the development of the application. |
| change capture Systems | Databases have long had similar log mechanisms such as Golden Gate. However these mechanisms are limited to database changes only and are not a general purpose event capture platform. They tend to focus primarily on the replication between databases, often between instances of the same database system (e.g. Oracle-to-Oracle). |

# Needs and Solution Proposal (V)

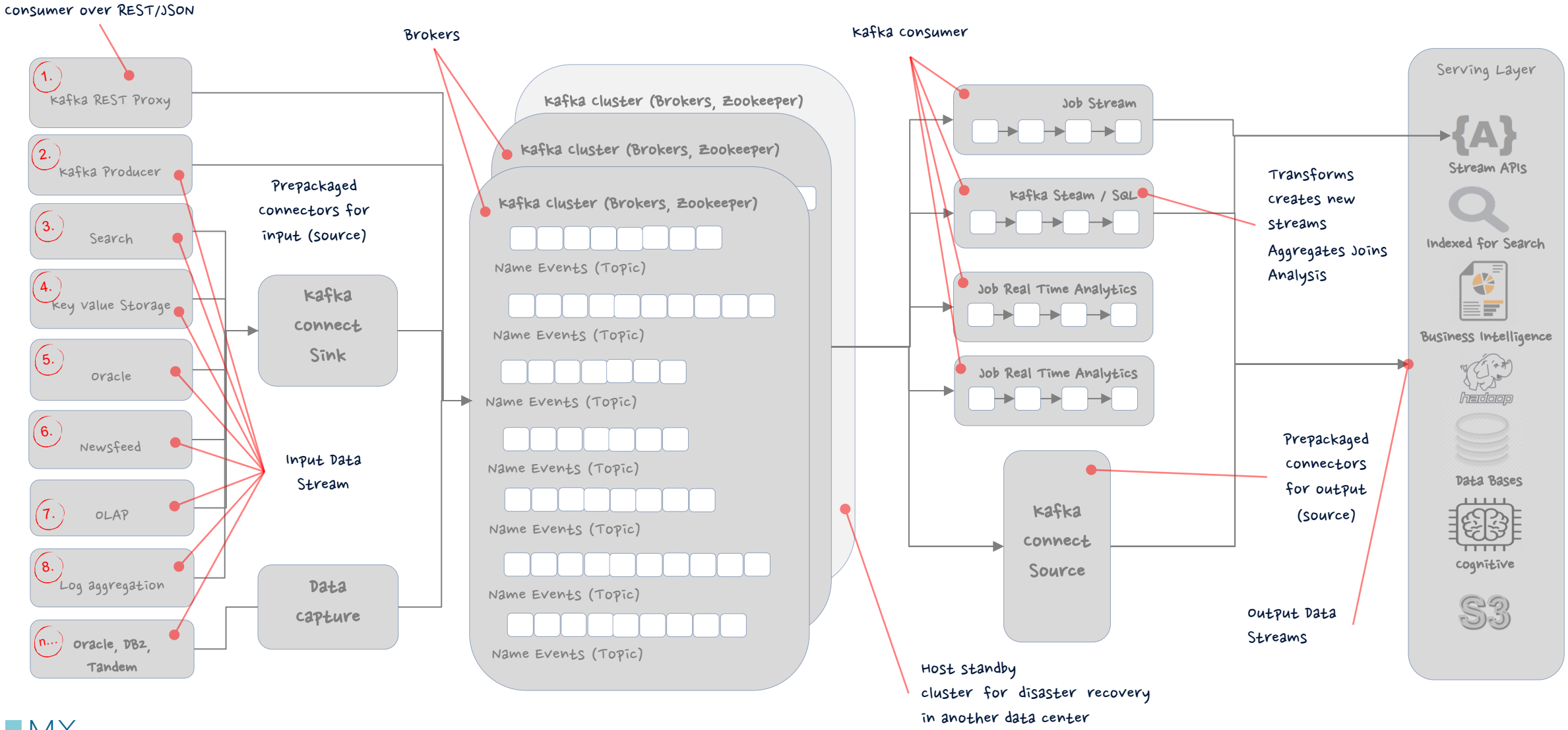
## Event Hub Architecture – The key capabilities of an Event Hub

Points to highlight about the relationship this event streaming platform concept has with other.

| System | Description |
|---|---|
| Data warehouses and Apache Hadoop | Event Hub does not replace data warehouse; It acts as a conduit for data to flow quickly for long-term retention, ad hoc analysis and batch processing. That same channel can be executed in reverse to publish the results derived from the processing by night or hourly batches. |
| Stream Processing Systems | External flow processing structures, such as Storm, Samza, Flink or Spark Streaming, can be used to provide the capacity of a transmission platform. They try to add additional processing for transformations in real time |

# Needs and Solution Proposal (VI)

## Event Hub Architecture – Kafka Connect



# Needs and Solution Proposal (VII)

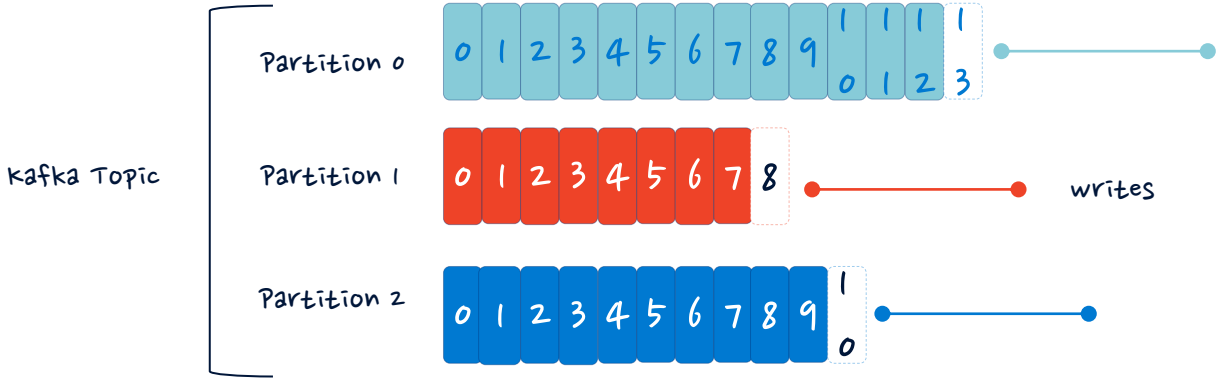
## Event Hub Architecture – Topics, Partitions and Offsets

Topics: a particular stream of data

- Similar to a table in database (without all the constraints)
- It have as many topics as you want
- A Topic is identified by its name

Topics are split in partitions

- Each partition is ordered
- Each massage within a partition gets an incremental id, called offset

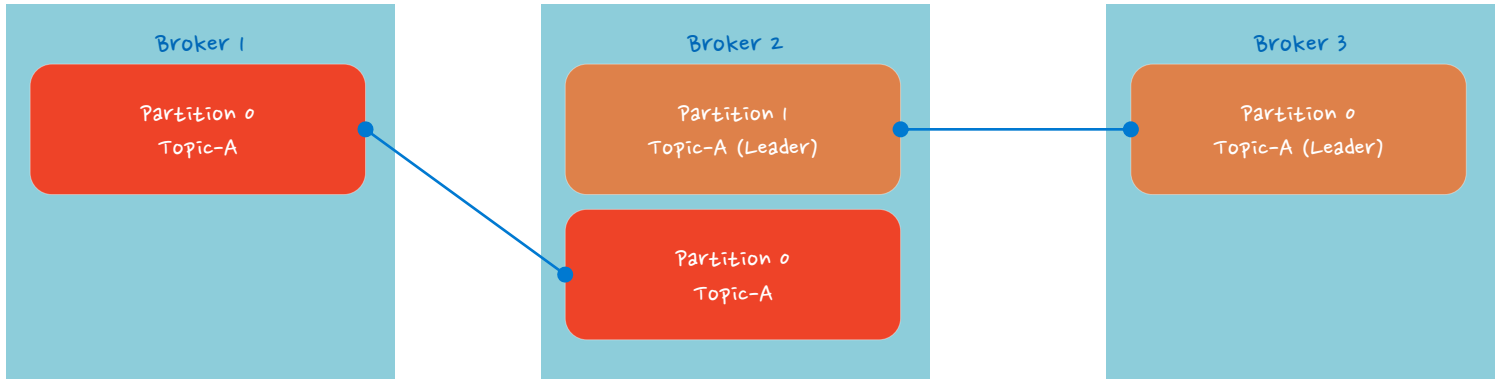



- Offset only have a meaning for a specific partition
- Order is guaranteed only within a partition (not across partitions)
- Data is kept only for a limited time
- Once the data is written to partition, it can't be changed (immutability)
- Data is assigned randomly to a partition unless a key is provided

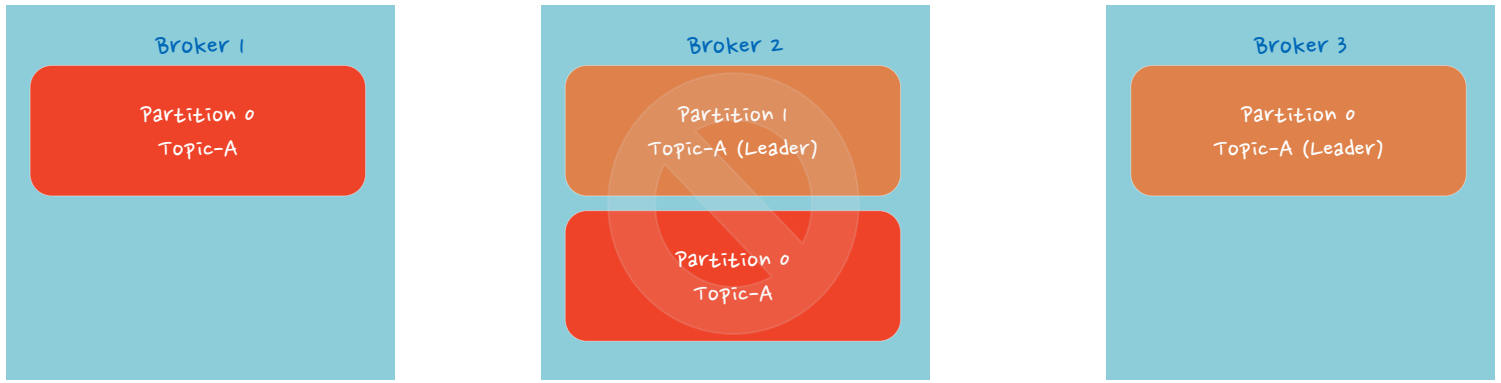
# Needs and Solution Proposal (VIII)

## Event Hub Architecture – Topics Replication factor

- Topic should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: Topic-A with 2 partition and replication factor of 2



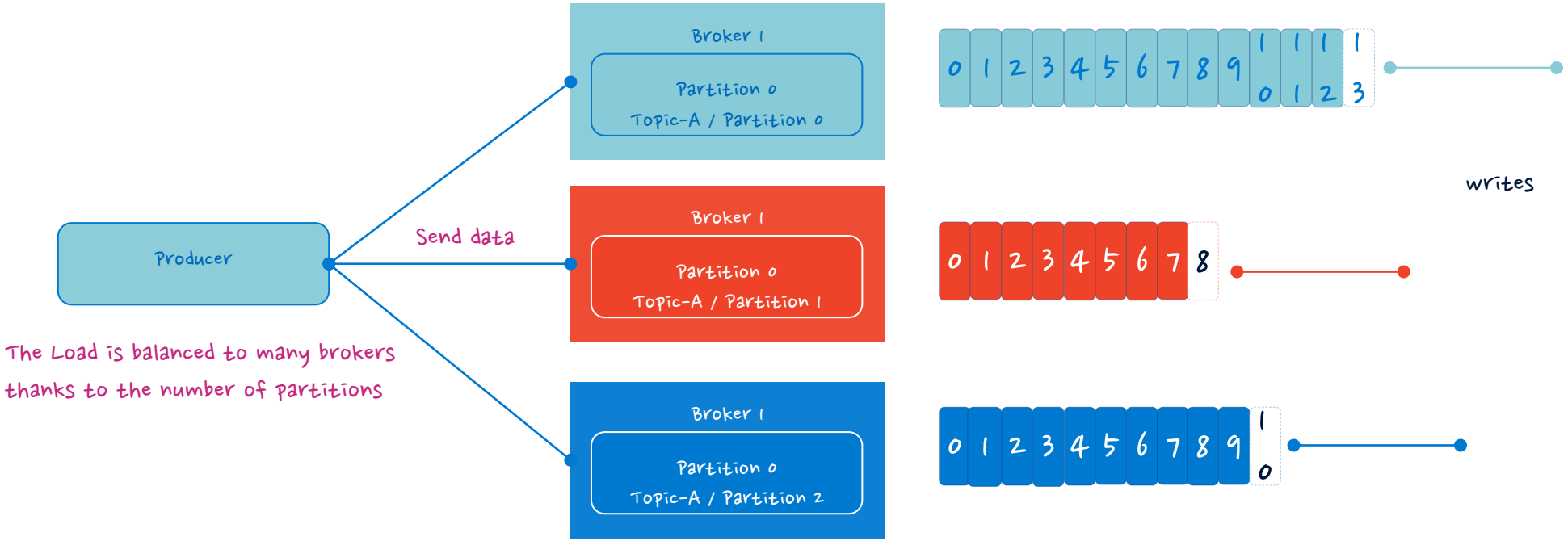

- Example lost Broker 2
- Result: Broker 1 and 3 can still serve the data



# Needs and Solution Proposal (IX)

## Event Hub Architecture – Producer and Massage Keys

- Producer write data to topic (which is made of partitions)
- Producer automatically know to which broker and partition to write to
- In case of Broker failures, Producer will automatically recover
- Producer can choose to replicas acknowledgment of data writes:
  - acks=0: Producer won't wait acknowledgement (possible data loss)
  - acks=1: Producer won't wait acknowledgement (limited data loss)
  - acks=all: Producer won't wait acknowledgement (possible data loss)



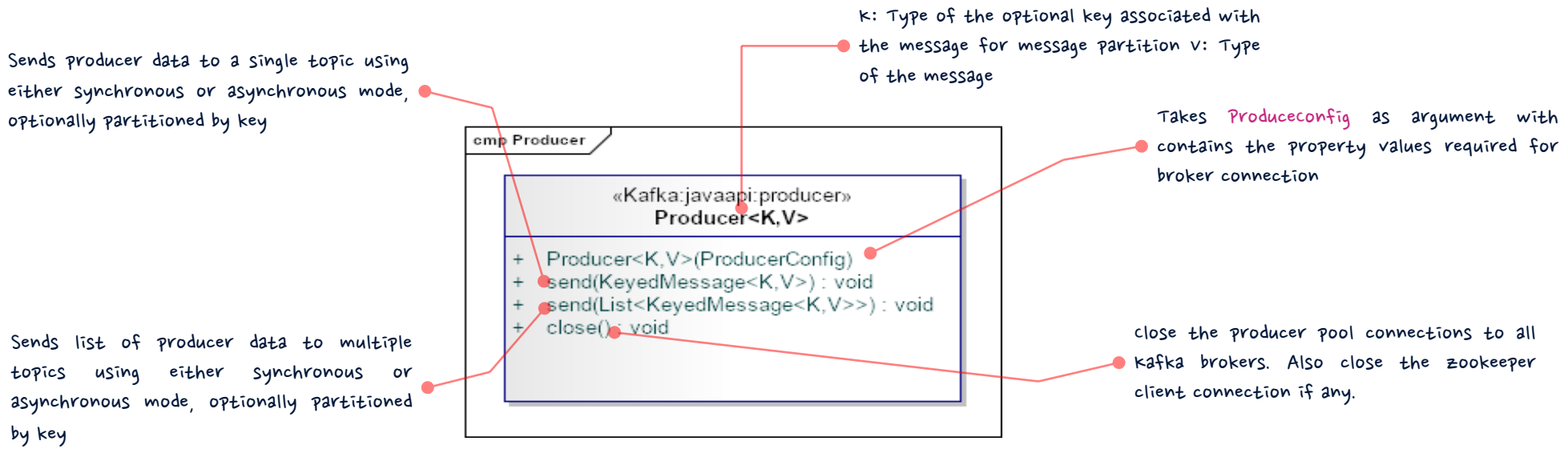
# Needs and Solution Proposal (X)

## Event Hub Architecture – Producer

Producers are applications that create messages and publish them to the kafka broker for further consumption. These producers can be different in nature; for example, frontend applications, backend services, proxy applications, adapters to legacy systems, and producers for Hadoop. These producers can also be implemented in different languages such as Java, Scala, C, and Python.

The Kafka API for message producers

For high efficiency in kafka, producers can also publish the messages in batches that work in asynchronous mode only. In asynchronous mode, the producer works either with a fixed number of messages or fixed latency defined by producer configuration, queue.time or batch.size, respectively.



# Needs and Solution Proposal (XI)

## Event Hub Architecture – Producer

### Defining properties

```
Properties props = new Properties();
props.put("metadata.broker.list"," name_server :9092, name_server:9093, name_server :9094");
props.put("serializer.class","kafka.serializer.StringEncoder");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<String, String> producer = new Producer<String, String> (config);
```

### Building the message and sending it

```
String runtime = new Date().toString();;
String msg = "Message Publishing Time - " + runtime;
KeyedMessage<String, String> data = new KeyedMessage<String, String> (topic, msg);
producer.send(data);
```

metadata.broker.list: this property specifies the list of intermediaries (in the format [<node: port>, <node: port>]) to which the producer must connect. Kafka producers automatically determine the leading broker for the subject, divide it by means of a metadata request and connect to the correct broker before publishing any message.

serializer.class: This property specifies the serializer class that needs to be used while preparing the message for transmission from the producer to the broker. In this example, we will be using the string encoder provided by Kafka. By default, the serializer class for the key and message is the same, but we can also implement the custom serializer class by extending the Scala-based kafka.serializer.Encoder implementation. Producer configuration key.serializer.class is used to set the custom encoder.

request.required.acks: This property instructs the Kafka broker to send an acknowledgment to the producer when a message is received. The value 1 means the producer receives an acknowledgment once the lead replica has received the data. This option provides better durability as the producer waits until the broker acknowledges the request as successful. By default, the producer works in the "fire and forget" mode and is not informed in the case of message loss.

# Needs and Solution Proposal (XII)

## Event Hub Architecture – Producer (Simple Java Producers)

Set the broker list for requesting metadata to find the lead broker.

This specifies the serializer class for keys.

1 means the producer receives an acknowledgment once the lead replica has received the data. This option provides better durability as the client waits until the server acknowledges the request as successful.

Topic name and the message count to be published is passed from the command line.

creates a KeyedMessage instance

Publish the message

creates a KeyedMessage instance

```
package mx.citi.bigdata.common;

import java.util.Date;
import java.util.Properties;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class SimpleProducer {
  private static Producer<String, String> producer;
  public SimpleProducer() {
    Properties props = new Properties();
    props.put("metadata.broker.list",
            "197.168.128.132:9092, 197.168.128.132:9093,
             197.168.128.132:9094");
    props.put("serializer.class", "kafka.serializer.StringEncoder");
    props.put("request.required.acks", "1");
    ProducerConfig config = new ProducerConfig(props);
    producer = new Producer<String, String>(config);
  }

  public static void main(String[] args) {
    int argsCount = args.length;
    if (argsCount == 0 || argsCount == 1)
      throw new IllegalArgumentException(
        "Please provide topic name and Message count as arguments");
    String topic = (String) args[0];
    String count = (String) args[1];
    int messageCount = Integer.parseInt(count);
    System.out.println("Topic Name - " + topic);
    System.out.println("Message Count - " + messageCount);
    SimpleProducer simpleProducer = new SimpleProducer();
    simpleProducer.publishMessage(topic, messageCount);
  }

  private void publishMessage(String topic, int messageCount) {
    for (int mCount = 0; mCount < messageCount; mCount++) {
      String runtime = new Date().toString();
      String msg = "Message Publishing Time - " + runtime;
      System.out.println(msg);
      KeyedMessage<String, String> data =
        new KeyedMessage<String, String>(topic, msg);
        producer.send(data);
    }
    producer.close();
  }
}
```

Note: This is an example, it is recommended the use of good practices at the design and language level in which the producer will be implemented

# Needs and Solution Proposal (XIII)

## Event Hub Architecture – Producer (Simple Scala Producers)

Set the broker list for requesting metadata to find the lead broker.

This specifies the serializer class for keys.

The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances.

creates a record to be sent to a specified topic and partition

Send the given record asynchronously and return a future which will eventually contain the response information.

close producer connection with broker

```
package mx.citi.bigdata.common;

import java.util.Properties
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
object KafkaProducerApp extends App {

  val props:Properties = new Properties()
  props.put("bootstrap.servers"197.168.128.132:9092,
            197.168.128.132:9093,
            197.168.128.132:9094")
  props.put("key.serializer",
         "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer",
         "org.apache.kafka.common.serialization.StringSerializer")
  props.put("acks","all")
  val producer = new KafkaProducer[String, String](props)
  val topic = "text_topic"
  try {
    for (i <- 0 to 15) {
      val record = new ProducerRecord[String, String](topic, i.toString, "Produce" + i)
      val metadata = producer.send(record)
      printf(s"Sent Record(key=%s value=%s) " +
        "meta(partition=%d, offset=%d)\n",
        record.key(), record.value(),
        metadata.get().partition(),
        metadata.get().offset())
    }
  }catch{
    case e:Exception => e.printStackTrace()
  }finally {
    producer.close()
  }
}
```

Note: This is an example, it is recommended the use of good practices at the design and language level in which the producer will be implemented

# Needs and Solution Proposal (XIV)

## Event Hub Architecture – Consumers & Consumer Groups

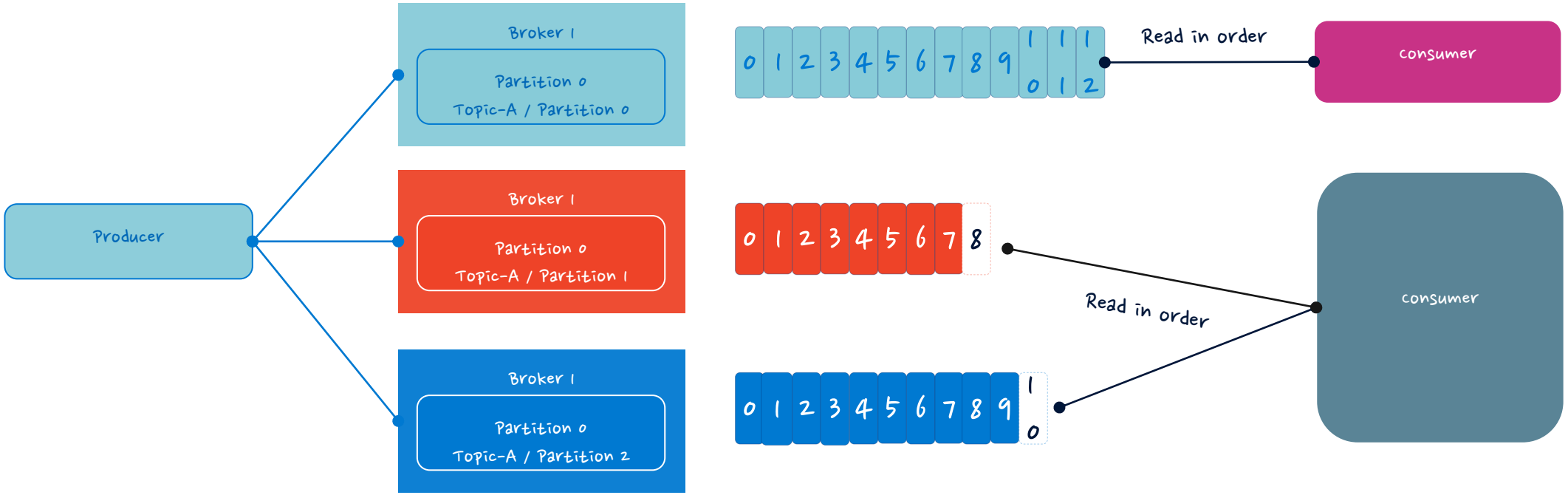
consumer

consumer read data from topic (identified by name)

consumer know with broker to read from

In case of broker failures, consumers know how to recover

Data is read in order within each partitions



# Needs and Solution Proposal (XV)

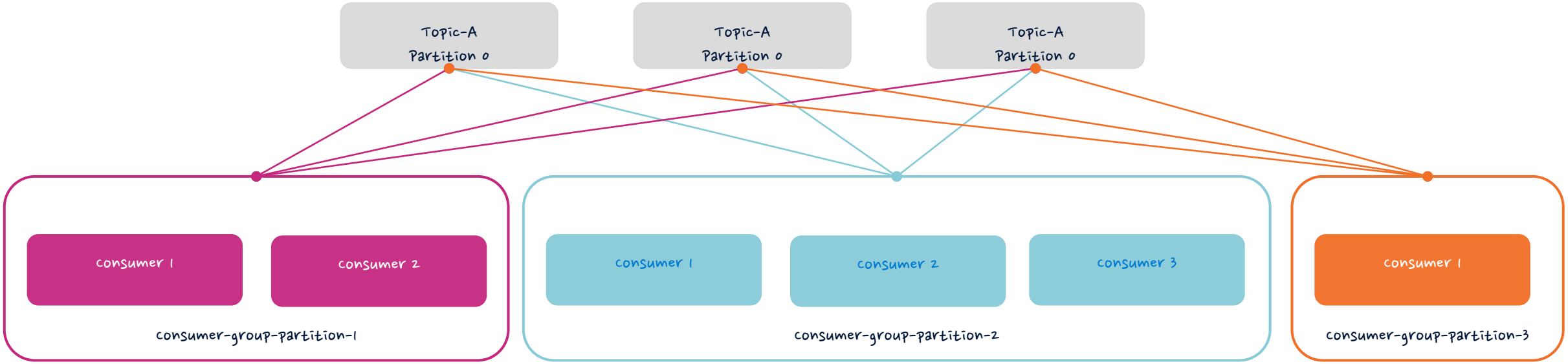
## Event Hub Architecture – Consumers & Consumer Groups

consumer Groups

consumer read data in consumer groups

Each consumer within a group read from exclusive partition

If you have more consumers than partition, some consumers will be inactive



# Needs and Solution Proposal (XVI)

## Event Hub Architecture – Consumers Offsets

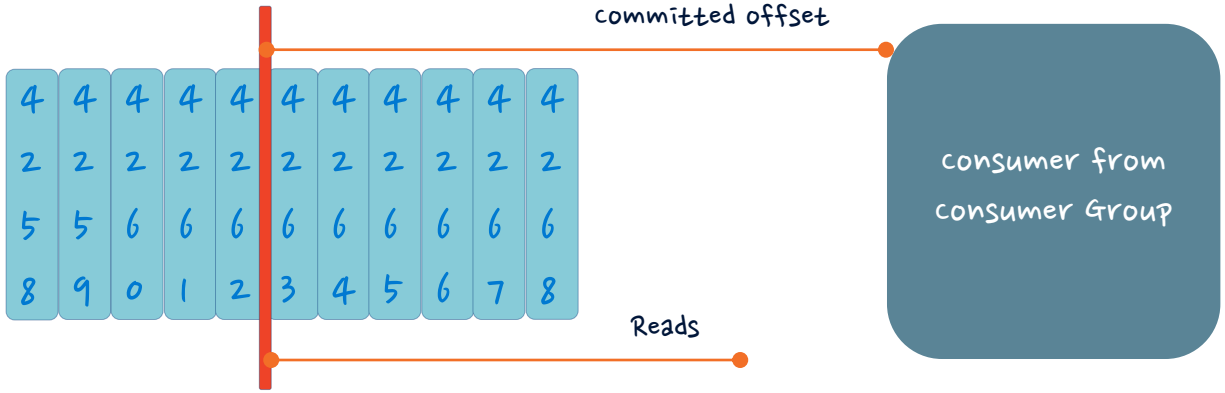
consumer Groups

kafka stores the offset at which a consumer group has been reading

The offsets committed live in kafka topic named __consumer_offsets

When a consumer in group has processed data received from kafka, it should be committing the offsets

If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offset



# Needs and Solution Proposal (XVII)

## Event Hub Architecture – Zookeeper

Zookeper

Zookeeper manager brokers (keeps a list of them)

Zookeeper help in performing leader election for partition

Zookeeper send notification to kafka in case of change (new topic, broker comes up delete topics, etc.)

kafka can't work without Zookeeper

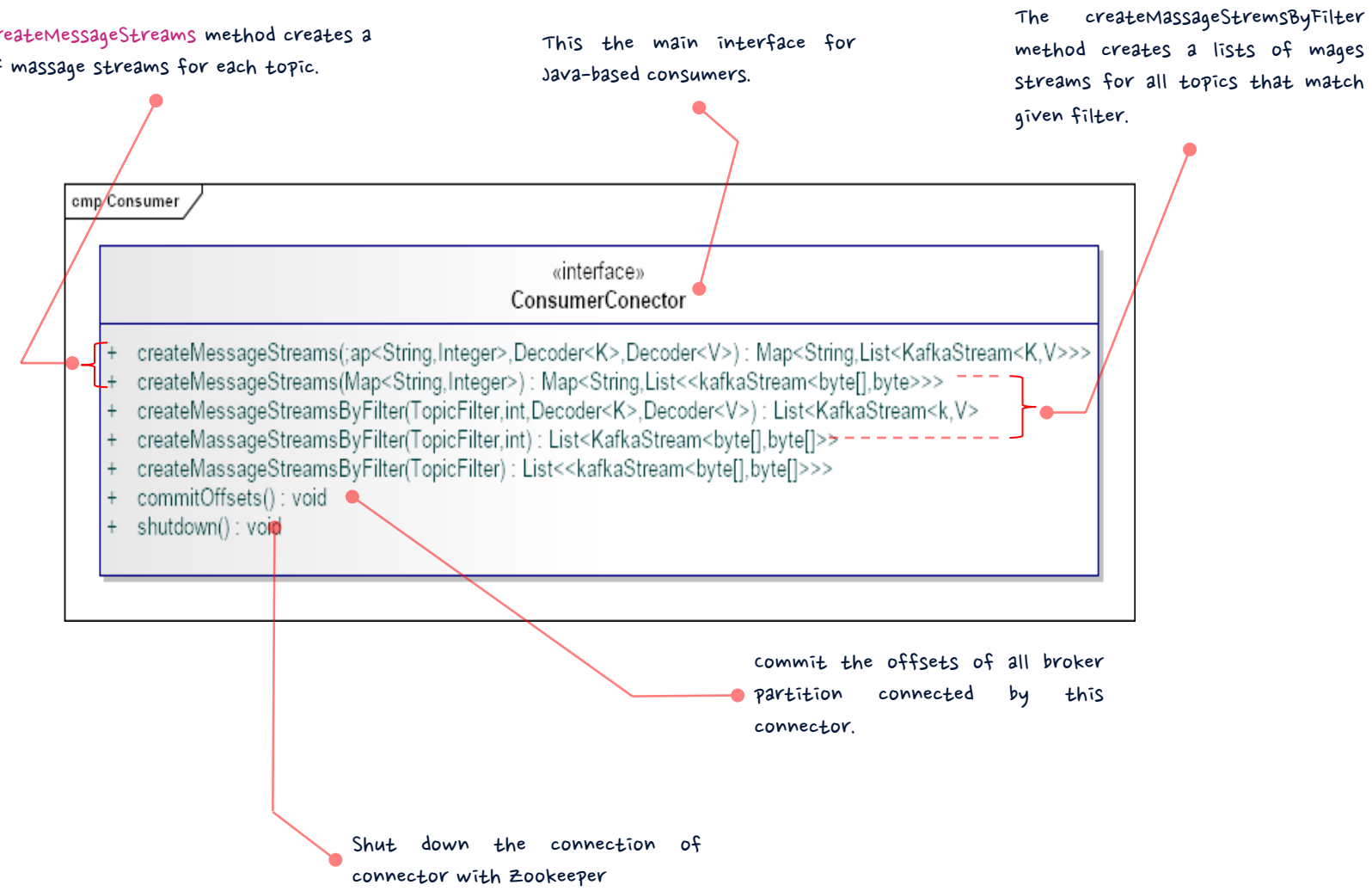
Zookeeper by design operates with an odd number of server (3,5,7)

Zookeeper has a leader (handle writes) the rest of the server are followers (handle reads)

# Needs and Solution Proposal (XVIII)

## Event Hub Architecture – Consumers High-level API

consumerconnector: Kafka provides the consumerconnector interface (interface consumerconnector) that is further implemented by the Zookeeperconsumerconnector class (kafka.javaapi.consumer.Zookeeperconsumerconnector). This class is responsible for all the interaction a consumer has with Zookeeper.



# Needs and Solution Proposal (XIX)

## Event Hub Architecture – Consumers High-level API

Defining properties

```
class KafkaStream[K,V](private val queue:
                       BlockingQueue[FetchedDataChunk],
                       consumerTimeoutMs: Int,
                       private val keyDecoder: Decoder[K],
                       private val valueDecoder: Decoder[V],
                       val clientId: String)
```

kafkaStream: objects of the kafka.consumer.kafkaStream class are returned by the createMessageStreams call from the consumerconnector implementation. This list of the kafkaStream objects is returned for each topic, which can further create an iterator over messages in the stream.

The parameters K and V specify the type for the partition key and message value, respectively

The low-level consumer API

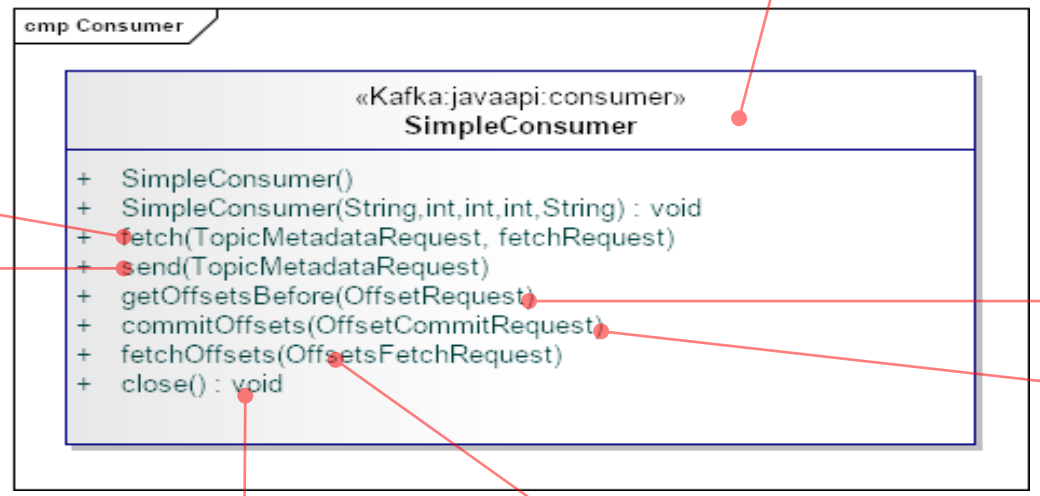
The high-level API does not allow consumers to control interactions with brokers. Also known as "simple consumer API", the low-level consumer API is stateless and provides fine grained control over the communication between kafka broker and the consumer. It allows consumers to set the message offset with every request raised to the broker and maintains the metadata at the consumer's end. This API can be used by both online as well as offline consumers such as Hadoop. These types of consumers can also perform multiple reads for the same message or manage transactions to ensure the message is consumed only once.

compared to the high-level consumer API, developers need to put in extra effort to gain low-level control within consumers by keeping track of offsets, figuring out the lead broker for the topic and partition, handling lead broker changes, and so on.

In the low-level consumer API, consumers first query the live broker to find out the details about the lead broker. Information about the live broker can be passed on to the consumers either using a properties file or from the command line. The topicsMetadata() method of the kafka.javaapi.TopicMetadataResponse class is used to find out metadata about the topic of interest from the lead broker. For message partition reading, the kafka.api.OffsetRequest class defines two constants: EarliestTime and LatestTime, to find the beginning of the data in the logs and the new messages stream. These constants also help consumers to track which messages are already read.

# Needs and Solution Proposal (XX)

## Event Hub Architecture – Consumers Low-level API




This method returns the set of messages from topic. Here FetchRequest specifies the topic name, topic partition, strating byte offset, and maximus byte to be fetched.

consumer class for kafka massages, accepts lead broker, broker port, connection timeout, buffer size, and client ID.

The method returns the list of valid offsets before the given time.

The method commits the offsets for a topic passed as OffsetcommitRequest.

This method returns the set of messages from topic. Here FetchRequest specifies the topic name, topic partition, strating byte offset, and maximus byte to be fetched.

This method returns the offsets for a topic passed as offsetFetchRequest.

Shut down the connection of connector with Zookeeper

# Needs and Solution Proposal (XXI)

## Event Hub Architecture – Consumers Low-level API

### Simple Java consumers

This is a single-threaded simple Java consumer developed using the high-level consumer API for consuming the messages from a topic.

### Defining properties

Properties for making a connection with Zookeeper and pass these properties to the Kafka consumer

**zookeeper.connect:** This property specifies the Zookeeper <node:port> connection detail that is used to find the Zookeeper running instance in the cluster. In the Kafka cluster, Zookeeper is used to store offsets of messages consumed for a specific topic and partition by this consumer group.

```
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("group.id", "testgroup");
props.put("zookeeper.session.timeout.ms", "500");
props.put("zookeeper.sync.time.ms", "250");
props.put("auto.commit.interval.ms", "1000");
new ConsumerConfig(props);
```

**group.id:** This property specifies the name for the consumer group shared by all the consumers within the group. This is also the process name used by Zookeeper to store offsets.

**zookeeper.session.timeout.ms:** This property specifies the Zookeeper session timeout in milliseconds and represents the amount of time Kafka will wait for Zookeeper to respond to a request before giving up and continuing to consume messages.

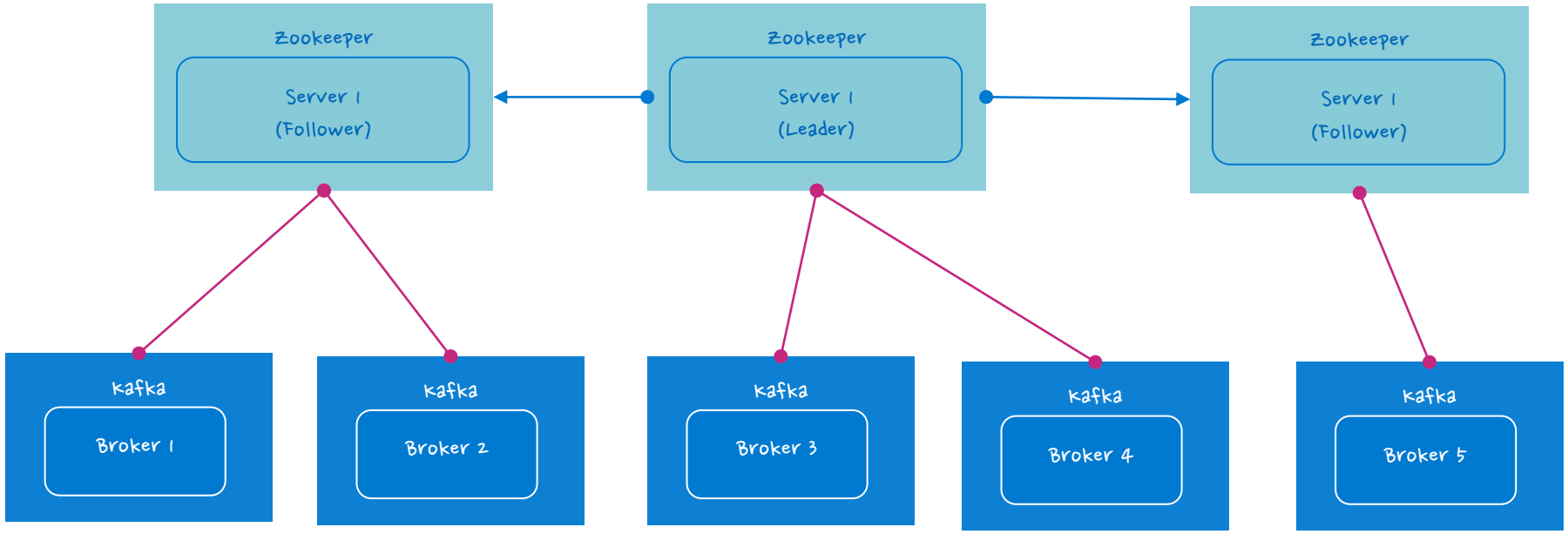
**auto.commit.interval.ms:** This property defines the frequency in milliseconds at which consumer offsets get committed to Zookeeper.

**zookeeper.sync.time.ms:** This property specifies the Zookeeper sync time in milliseconds between the Zookeeper leader and the followers.

Note: This is an example, it is recommended the use of good practices at the design and language level in which the producer will be implemented

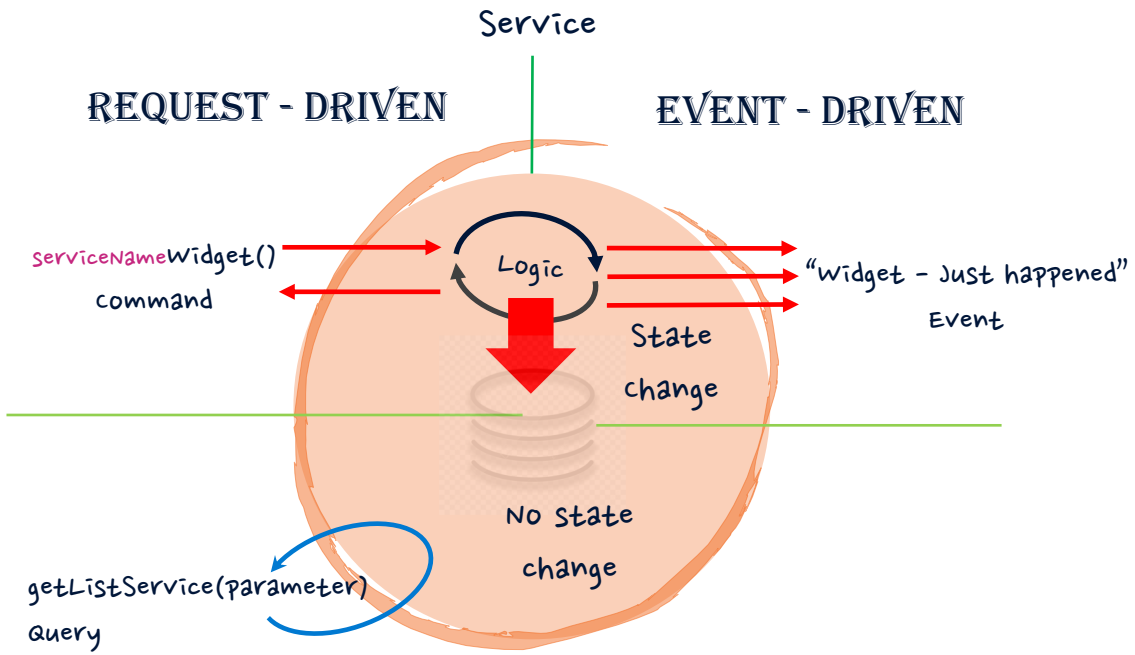
# Needs and Solution Proposal (XXII)

## Event Hub Architecture – Zookeeper



# Needs and Solution Proposal (XXIII)

## Event Hub Architecture – Even Driver




commands, events, and queries

The three mechanisms through which services interact

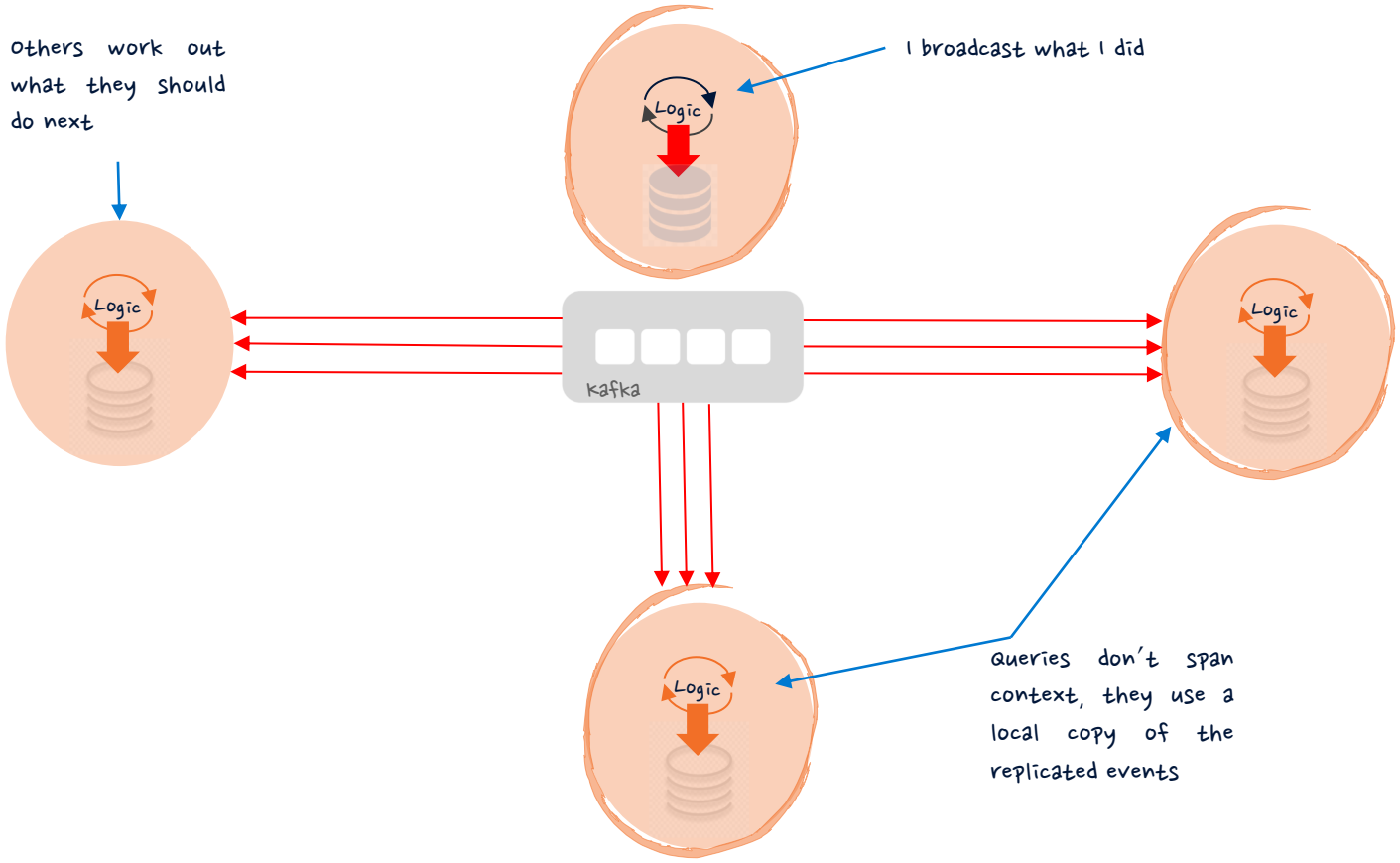
commands are actions—requests for some operation to be performed by another service, something that will change the state of the system. commands execute synchronously and typically indicate completion, although they may also include a result.
**Example:** processPayment(), returning whether the payment succeeded.
**When to use:** On operations that must complete synchronously, or when using orchestration or a process manager. consider restricting the use of commands to inside a bounded context.

Events are both a fact and a notification. They represent something that happened in the real world but include no expectation of any future action. They travel in only one direction and expect no response (sometimes called "fire and forget"), but one may be "synthesized" from a subsequent event.
**Example:** ordercreated{widget}, customerDetailsUpdated{customer}
**When to use:** When loose coupling is important (e.g., in multiteam systems), where the event stream is useful to more than one service, or where data must be replicated from one application to another. Events also lend themselves to concurrent execution.

Queries are a request to look something up. Unlike events or commands, queries are free of side effects; they leave the state of the system unchanged.
**Example:** getOrder(ID=42) returns order(42,···)
**When to use:** For lightweight data retrieval across service boundaries, or heavyweight data retrieval within service boundaries.

# Needs and Solution Proposal (XXIV)

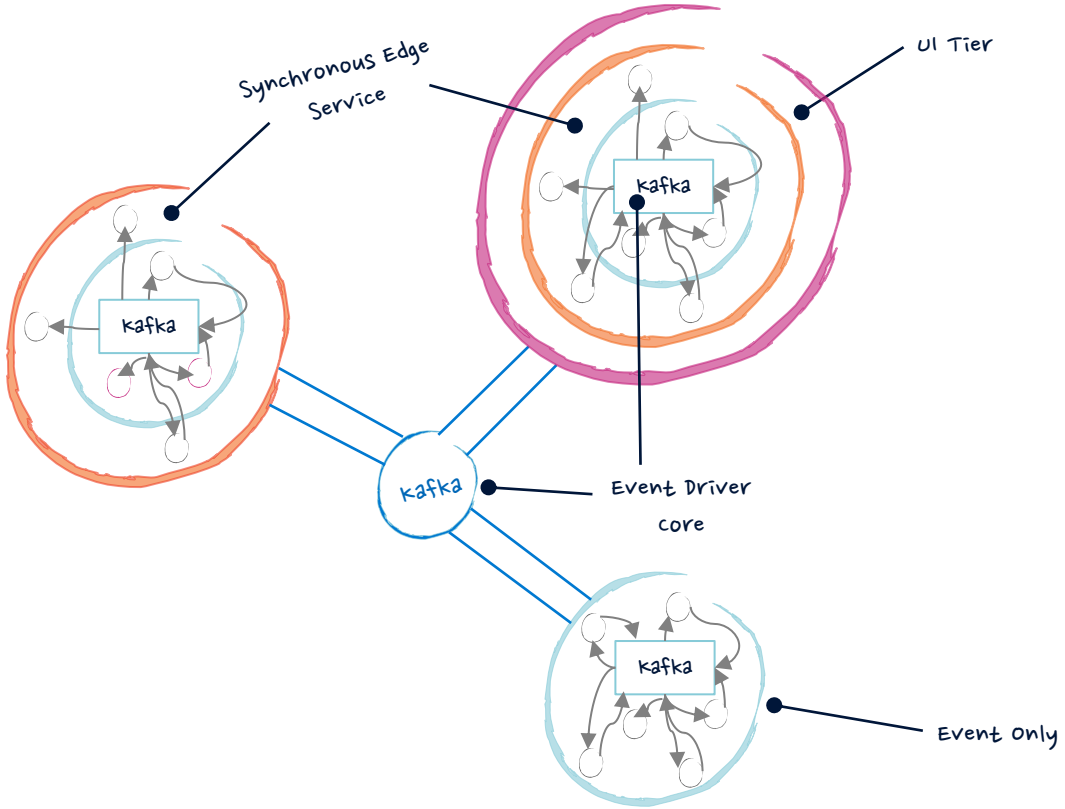
## Event Hub Architecture – Even Driver




Event-driven approaches demonstrating how event-driven approaches provide less coupling

# Needs and Solution Proposal (XVII)

## Event Hub Architecture – Even Driver Core



# Needs and Solution Proposal (XVIII)

## Event Hub Architecture – Even Producer GoldenGate

# Needs and Solution Proposal (XIX)

## Event Hub Architecture – Even Processor

# Needs and Solution Proposal (XX)

## Event Hub Architecture – Event Driven Services

# Infrastructure and Connectivity (I)

## Resumen de Componentes de Infraestructura

| Infrastructure component | Description | Applicability (New, Modified, Unmodified) | Application in which the component is located |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Volumetría (I)

## Case of use -

| Characteristic | Detalle |
|---|---|
| | |
| | |
| | |
| | |
| | |

# Volumetría (II)

## Case of use -

| Characteristic | Detalle |
|---|---|
| | |
| | |
| | |
| | |
| | |

# Volumetría (III)

## Case of use -

| Characteristic | Detalle |
|---|---|
| | |
| | |
| | |
| | |
| | |

# Servicios Generales (I)

## Seguridad

El detalle de los lineamientos mencionados están definidos en los documentos:

- Estándares

Confidencialidad

- Comunicación por medio de autenticación mutua entre Teredarta y Data Lake

Auditoria:

- Generación de Logs.

# Risks and Dependencies (I)

## Coexistence

El detalle de los lineamientos mencionados están definidos en los documentos:

- Validar si hay estándares de seguridad

## Riesgos

El detalle de los lineamientos mencionados están definidos en los documentos:

- Validar si hay estándares de seguridad

# Risks and Dependencies (II)

## Dependencias

El detalle de los lineamientos mencionados están definidos en los documentos:

- Estándar Controles de Seguridad para aplicativos de GFS México. (Respecto a los apartados de aplicaciones internas y controles de procesamiento)
- Estándar de Controles de Seguridad. Versión 1.0 Riesgo

# Standards, Patterns and Decisions of Architecture (I)

## Exceptions to Standards and Architecture Patterns

- …

# Standards, Patterns and Decisions of Architecture (II)

## Architecture Decisions

| Decisión | Premisas | Razonamiento de la Decisión |
|---|---|---|
| You opt for a local solution | ▪ The solution proposes an application developed in Scala, Spark or PySpark | |
| | | |
| | | |

# Standards, Patterns and Decisions of Architecture (III)

## Dependencias

| Dependencia | Componente externo a la solución que impacta | Fecha estimada en la que se requiere disponible |
|---|---|---|
| No Aplica | No Aplica | No Aplica |
| | | |
| | | |

# References and Annex (I)

## References

| Nombre y version | Fecha | Comentarios | Rol/Departamento |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |