Emerging Paradigms for Big Data

Messaging systems

Apache Kafka

Fátima Leal

DCT DEPARTAMENTO CIÊNCIA E TECNOLOGIA

UPT UNIVERSIDADE PORTUCALENSE

Content

- Messaging systems
- Apache Kafka
- Hands-on Activity

Messaging systems

Introduction

- Nowadays, **real-time information** is continuously **generated**.
- This data needs easy ways to be delivered to **multiple types of receivers**.
- Information generators and information consumers are inaccessible to each other; this is when **integration tools** enter the scene.

Understanding messaging systems

- Applications consuming data from one or more data sources.

- **Messaging systems** can be used as an **integration channel** for information exchange between different applications

- In any application integration system design, there are a few important principles that should be kept in mind:
  - loose coupling
  - common interface definitions,
  - latency, and reliability.

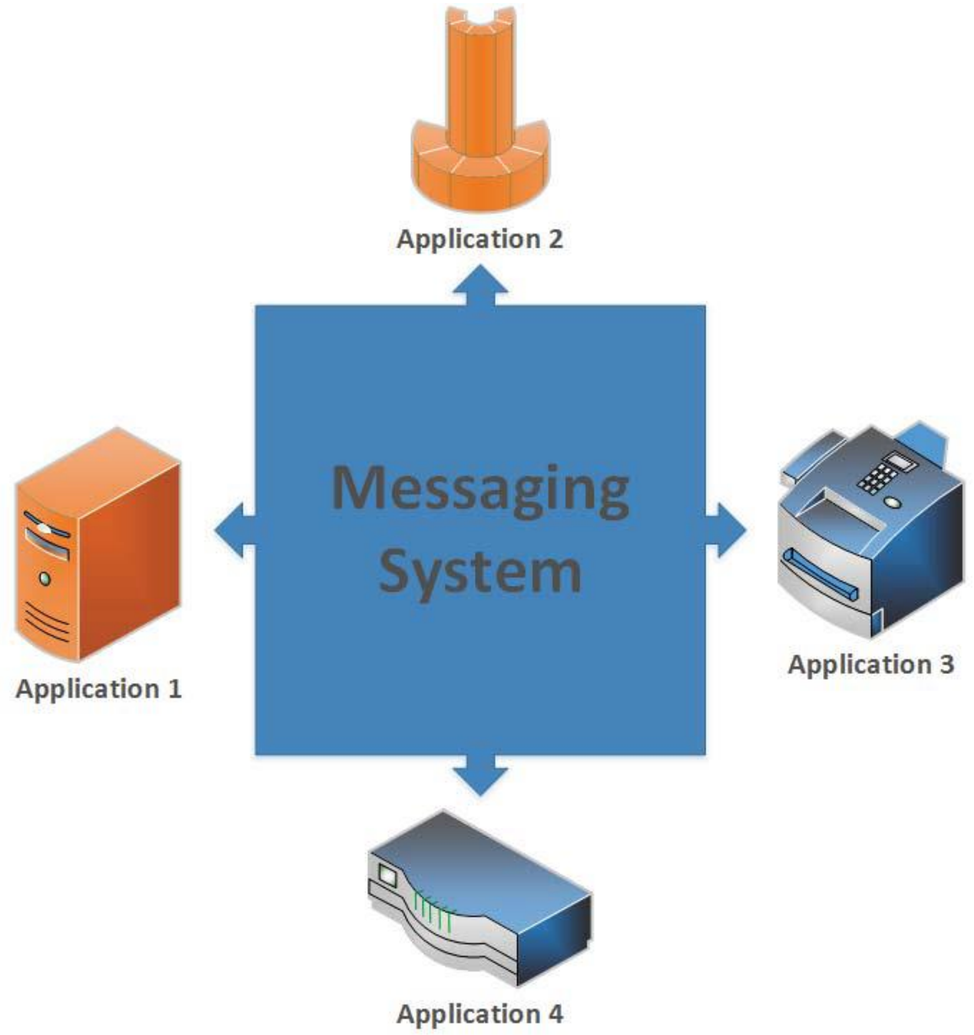
Understanding messaging systems

- **Loose coupling** between applications ensures minimal dependencies on each other.
- **Common interface definitions** ensure a **common** agreed-upon **data format** for exchange between applications.
- **Latency** is the time taken by messages to traverse between the sender and receiver.
- **Reliability** ensures that temporary unavailability of applications does not affect dependent applications that need to exchange information.

Understanding messaging systems

- A **messaging system** is one of the **most used mechanisms** for information exchange in applications.
- The **other mechanisms** used to share information could be Remote Procedure Calls (RPC), file share, shared databases, and web service invocation.
- You can **share small packets** of data or data streams **with other applications** using messaging in a timely and real-time fashion

Understanding messaging systems



Understanding messaging systems

- Basic messaging concepts:
  - **Message queues** are connectors between sending and receiving applications.
  - **Messages (data packets).** A message is an atomic data packet that gets transmitted over a network to a message queue. The sender application breaks data into smaller data packets and wraps it as a message with protocol and header information

Understanding messaging systems

- **Sender (producer):** Sender or producer applications are the sources of data that needs to be sent to a certain destination. They establish connections to message queue endpoints and send data in smaller message packets adhering to common interface standards.

- **Receiver (consumer):** Receiver or consumer applications are the receivers of messages sent by the sender application. They either pull data from message queues or they receive data from messages queues through a persistent connection.

Understanding messaging systems

- **Data transmission protocols**: Data transmission protocols determine rules to govern message exchanges between applications. Different queuing systems use different data transmission protocols. It depends on the technical implementation of the messaging endpoints.

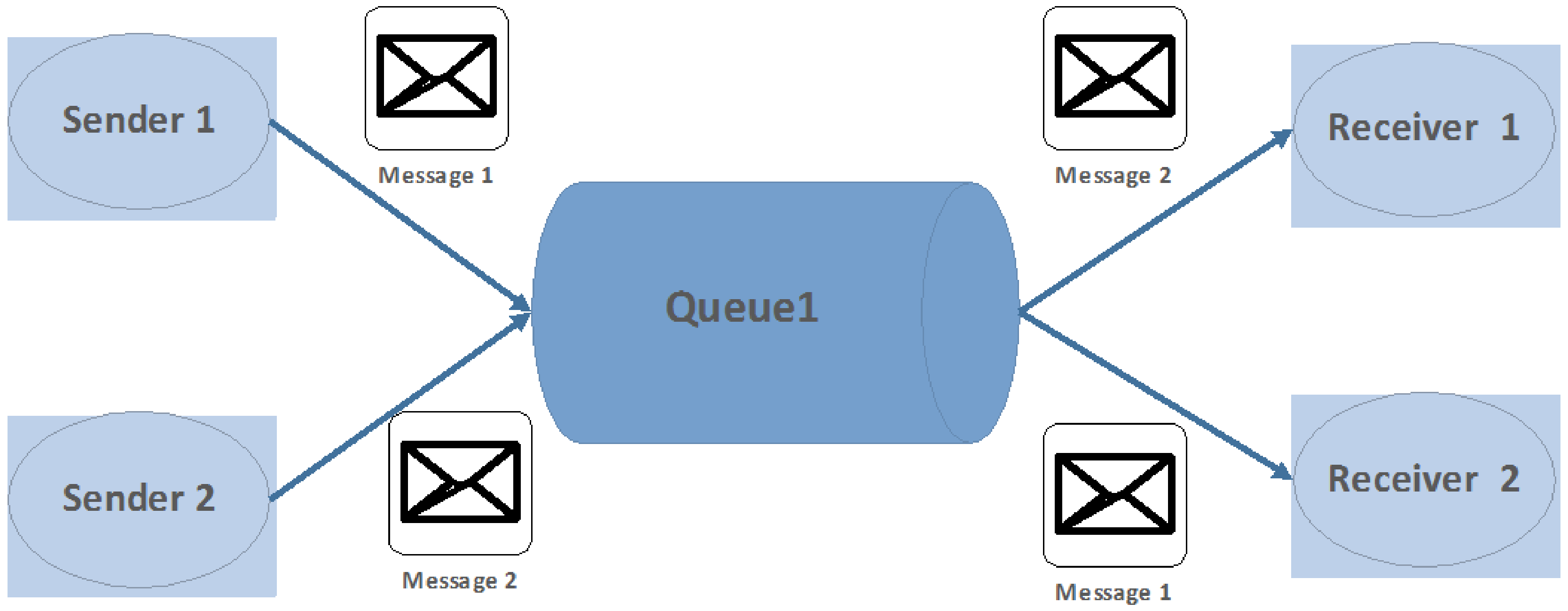
- **Transfer mode**: The transfer mode in a messaging system can be understood as the manner in which data is transferred from the source application to the receiver application.

Point-to-point messaging system

Point-to-point messaging system

- In a point-to-point (PTP) messaging model, **message producers** are called **senders** and **consumers** are called **receivers**.
- They **exchange messages** by means of a destination called a **queue**. Senders produce messages to a queue and receivers consume messages from this queue.
- What distinguishes point-to-point messaging is that a message can be consumed by only one consumer.

Point-to-point messaging system



Point-to-point messaging system

- Point-to-point messaging is generally used when **a single message** will be received by only **one message consumer**.

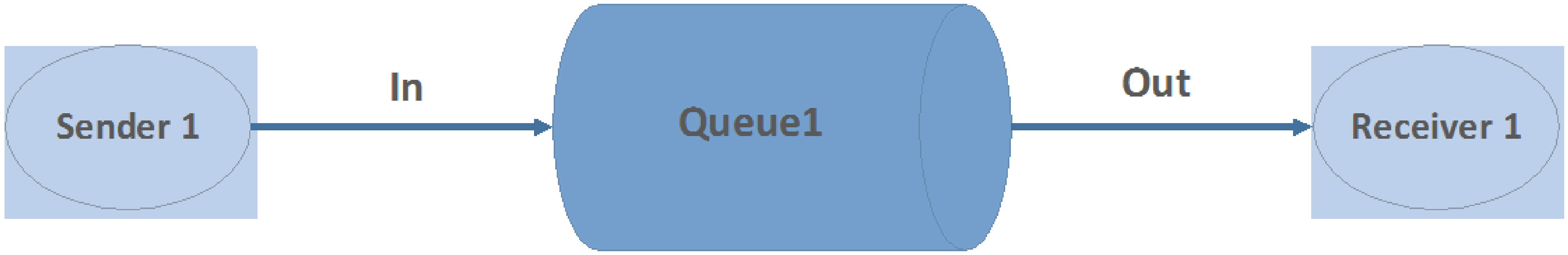
- There may be **multiple consumers listening** on the queue for the same message but **only one of the consumers** will receive it.

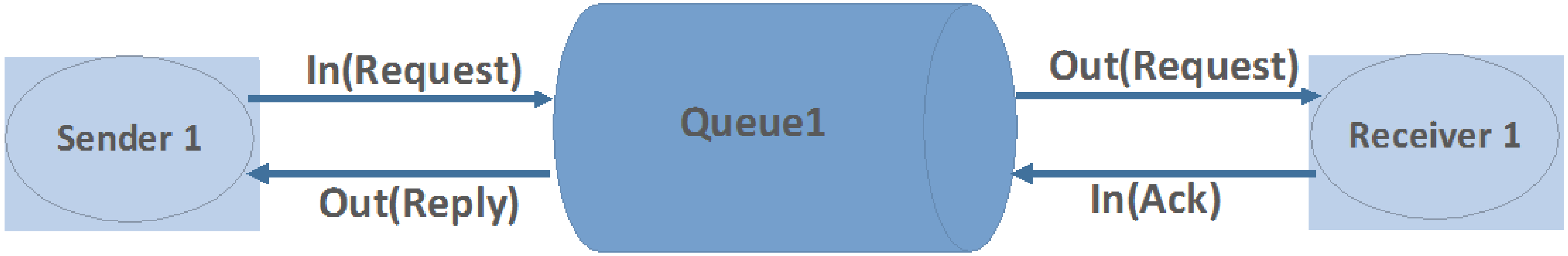
- They will be sending messages to the queue but it will be received by only one receiver.

Point-to-point messaging system

- The PTP messaging model can be further categorized into two types:
- In **fire-and-forget processing**, the **producer sends a message** to a centralized queue and **does not wait for any acknowledgment immediately**. It can be used in a **scenario** where you want to trigger an action or send a signal to the receiver to trigger some action that **does not require a response**.
- With an **asynchronous request/reply** PTP model, the message sender sends a message on one queue and then does a blocking wait on a reply queue **waiting for the response from the receiver**. The request/reply model provides for a high degree of decoupling between the sender and receiver, allowing the message producer and consumer components to be heterogeneous languages or platforms

Point-to-point messaging system





Publish-subscribe messaging system

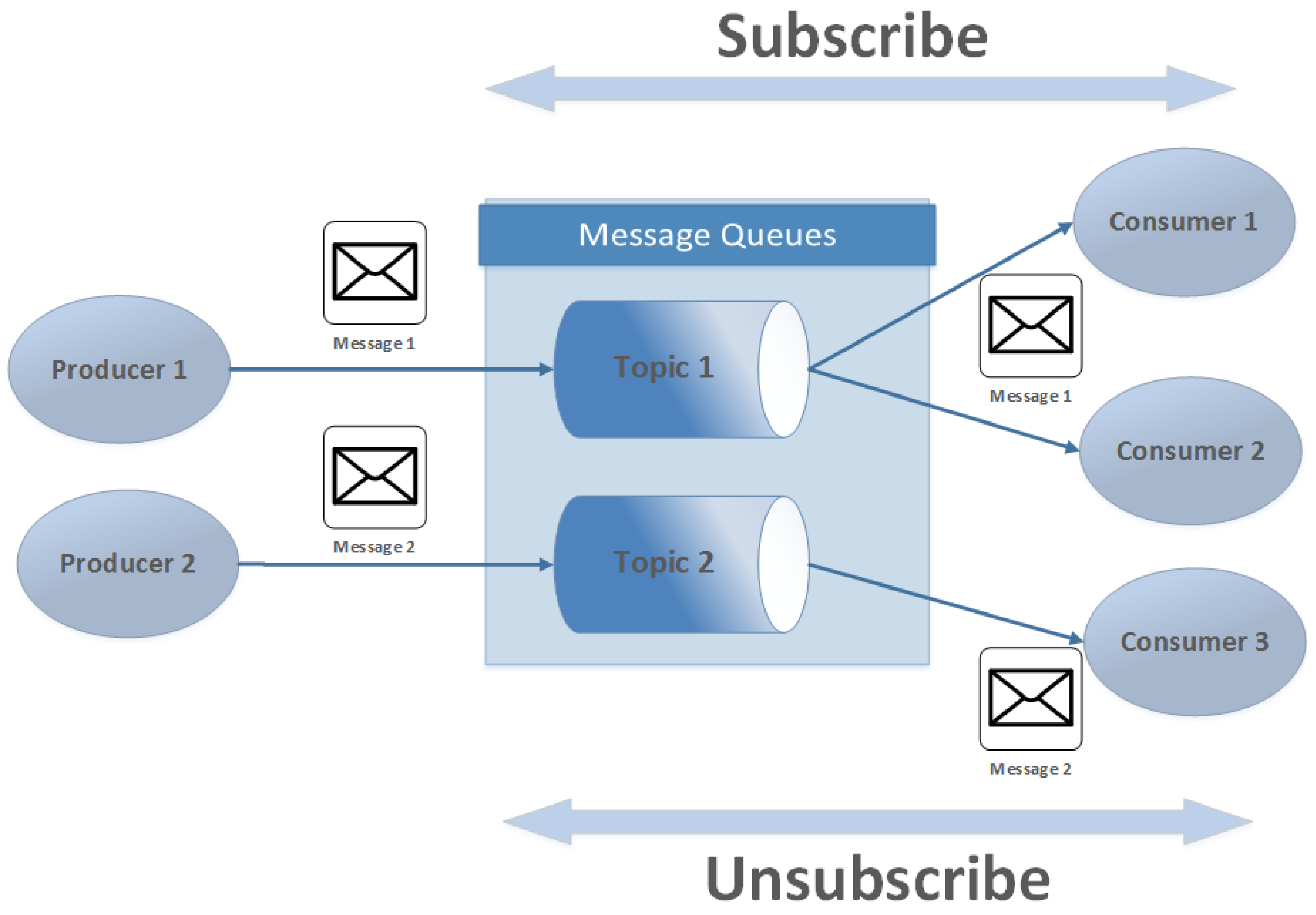
Publish-subscribe messaging system

- In a publish/subscribe (Pub/Sub) messaging model, a **subscriber registers its interest** in a particular topic or event and is subsequently **notified about the event asynchronously**

- Subscribers have the ability to express their interest in an event, or a pattern of events, and are subsequently notified of any event generated by a publisher that matches their registered interest.

- It is **different from the PTP** messaging model in a way that a topic can have multiple receivers and every receiver receives a copy of each message. In other words, a **message is broadcast** to all receivers without them having to poll the topic.

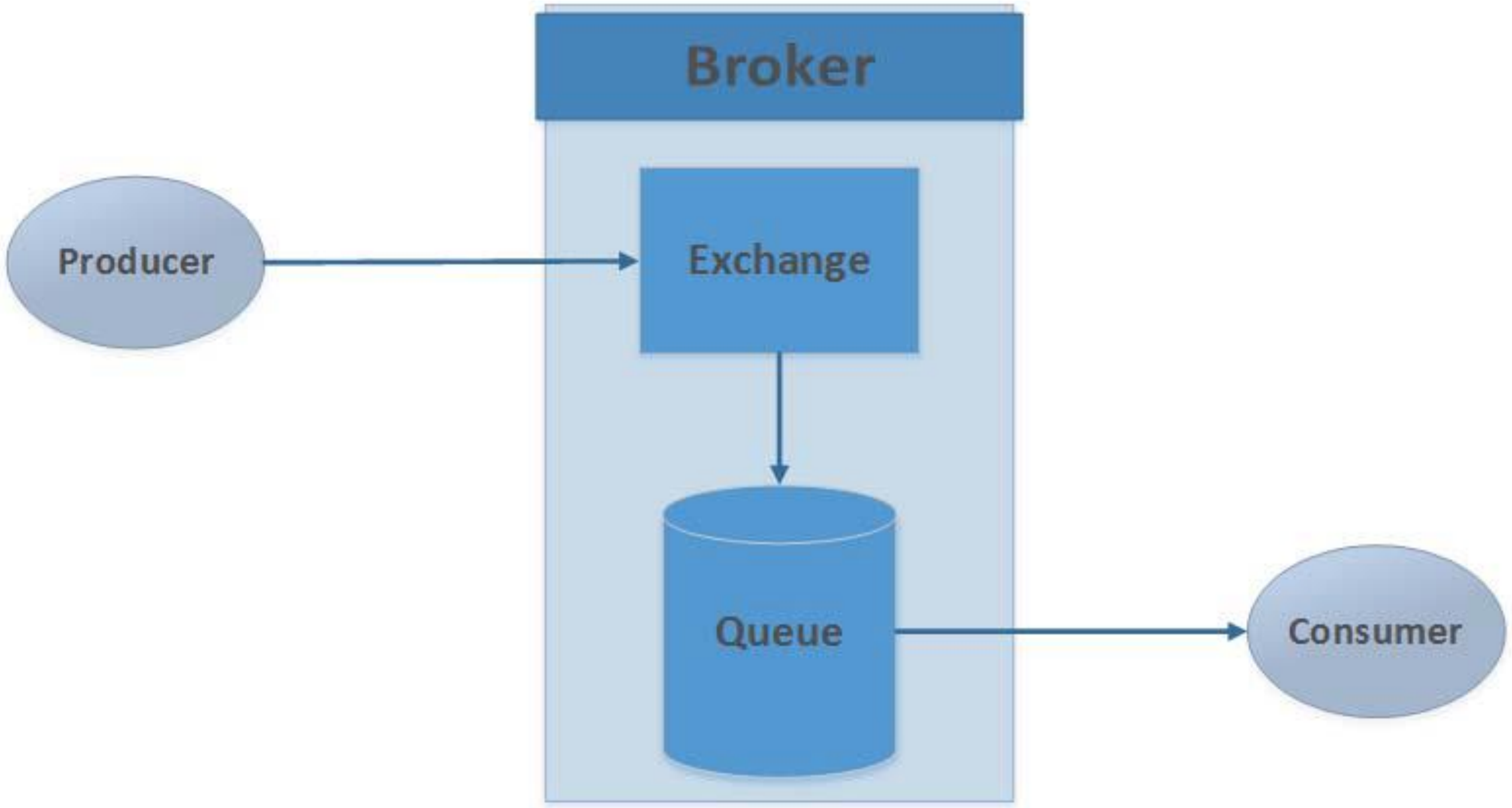
Publish-subscribe messaging system



Advance Queuing Messaging Protocol

- There are different data transmission protocols using which messages can be transmitted among sender, receiver, and message queues.
- Advance Message Queuing Protocol also known as AQMP is an **open protocol for asynchronous message queuing** that developed and matured over several years.
- AMQP provides richer sets of messaging functionalities that can be used to support very advanced messaging scenarios
- An AQMP messaging system consists of three main components: **Publisher(s), Consumer(s), Broker/server(s)**

Advance Queuing Messaging Protocol: Methods



Advance Queuing Messaging Protocol: Methods

- **Direct exchange** is a **key-based routing** mechanism. In this, a message is delivered to the queue whose **name** is equal to the **routing key** of the message.
- **Fan-out exchange** routes messages to all queues. If N queues are bound to a fan-out exchange, when a new message is published to that exchange, a copy of the message is delivered to all N queues.
- **Topic exchange** routes to some of the connected queues using **wildcards**. The topic exchange type is often used to implement various publish/subscribe pattern variations. Topic exchanges are commonly used for the multicast routing of messages.

Kafka

Introduction

- **Message publishing** is the mechanism for connecting heterogeneous applications through sending messages among them. The **message router** is known as **message broker**.

- **Apache Kafka** is a software solution to quickly **route real-time information to consumers**.

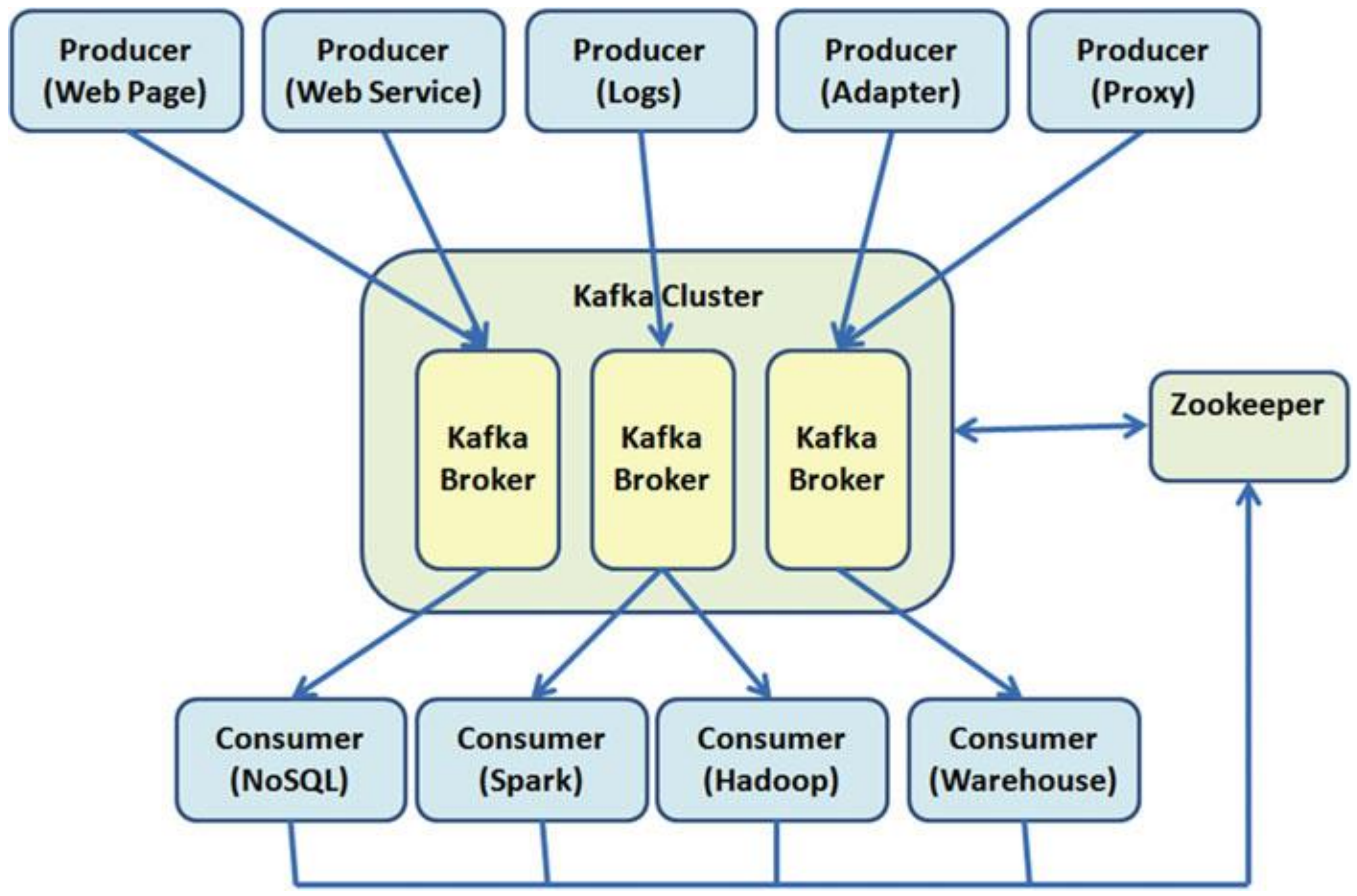
- The message broker provides integration:
  - Not blocking the producers
  - Producers do not know who the final consumers are

- **Apache Kafka is a real-time publish-subscribe solution messaging system**: open source, distributed, partitioned, replicated, commit-log based with a publish-subscribe schema.

Characteristics

- **Distributed**. Cluster-centric design that supports the distribution of the messages over the cluster members, maintaining the semantics.
- **Multiclient**. Easy integration with different clients from different platforms: Java, .NET, PHP, Ruby, Python, etc.
- **Persistent**. Kafka is designed with efficient O(1). *i.e.*, data structures provide constant time performance no matter the data size.
- **Real time**. The messages produced are immediately seen by consumer threads; these are the basis of the systems called complex event processing.
- **Very high throughput**. Kafka can handle hundreds of read and write operations per second from a large number of clients

Characteristics



Characteristics

- Actors on the **producers' side**:
  - Adapters. Generate transformation information.
  - Logs. The log files of application servers and other systems.
  - Proxies. Generate web analytics information.
  - Web pages. Front-end applications generating information.
  - Web services. The service layer; generate invocation traces.

- You could group the clients on the customer side:
  - **Offline** where the information is stored for posterior analysis.
  - **Near real time** where the information is stored but it is not requested at the same time.
  - **Real time where t**he information is analyzed as it is generated

Fast Data Era

- **Data is the new ingredient** of Internet-based systems
- Web page needs to know user activity, logins, page visits, clicks, scrolls, comments, heat zone analysis, shares, *etc*.
- Traditionally, the data was handled and stored with traditional aggregation solutions. Due to the high throughput, the analysis could not be done until the next day.
- Today, yesterday's information often useless.
- **Offline analysis** such as Hadoop is being left **out of the new economy**

Use Cases

- Commit logs - Uber, Spotify
- Log aggregation - Uber, Spotify
- Messaging
- Stream processing - LinkedIn, Netflix, Yahoo
- Record user activity

Kafka in Cluster

- Kafka, there are three types of clusters:
  - Single node–single broker
  - Single node–multiple broker
  - Multiple node–multiple broker

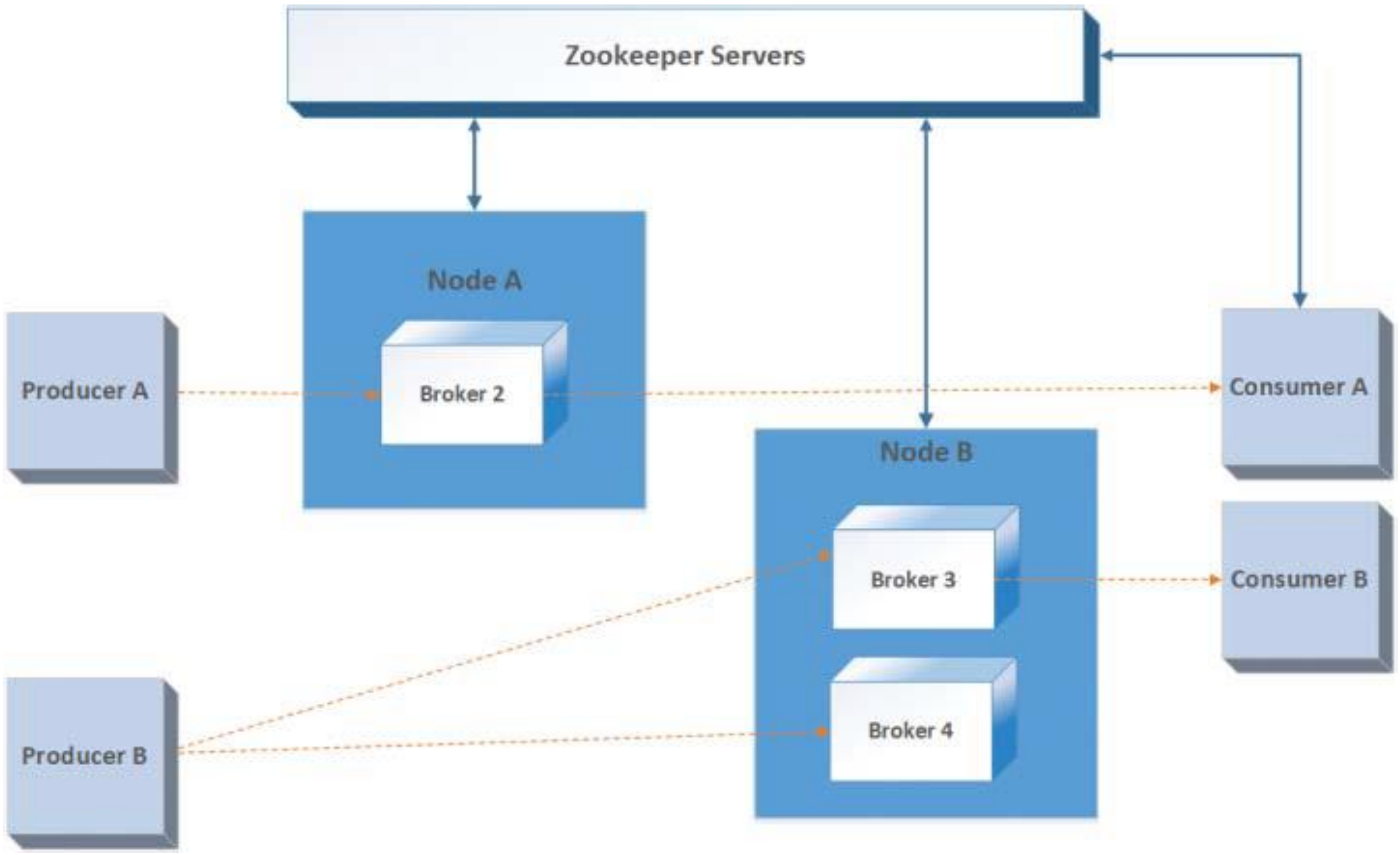
- A Kafka cluster has five main components:
  - **Topic**. A category or feed name in which messages are published by the message producers. **Topics are partitioned**; each partition is represented by an ordered immutable messages sequence.

  - **Broker**. A Kafka cluster has one or more physical servers in which each one may have one or more server processes running. Each server process is called a broker. The topics live in the broker processes.

Kafka in Cluster

- A Kafka cluster has five main components:
  - **Producer**. Publishes data to topics by choosing the appropriate partition in the topic. For load balancing, the messages allocation to the topic partition can be done in a round-robin mode or by defining a custom function.
  - **Consumer**. Applications or processes subscribed to topics and process the feed of published messages.
  - **ZooKeeper**. Coordinator between the broker and the consumers. ZooKeeper coordinates the distributed processes through a shared hierarchical name space of data registers; these registers are called znodes.

Kafka in Cluster



Kafka Architecture

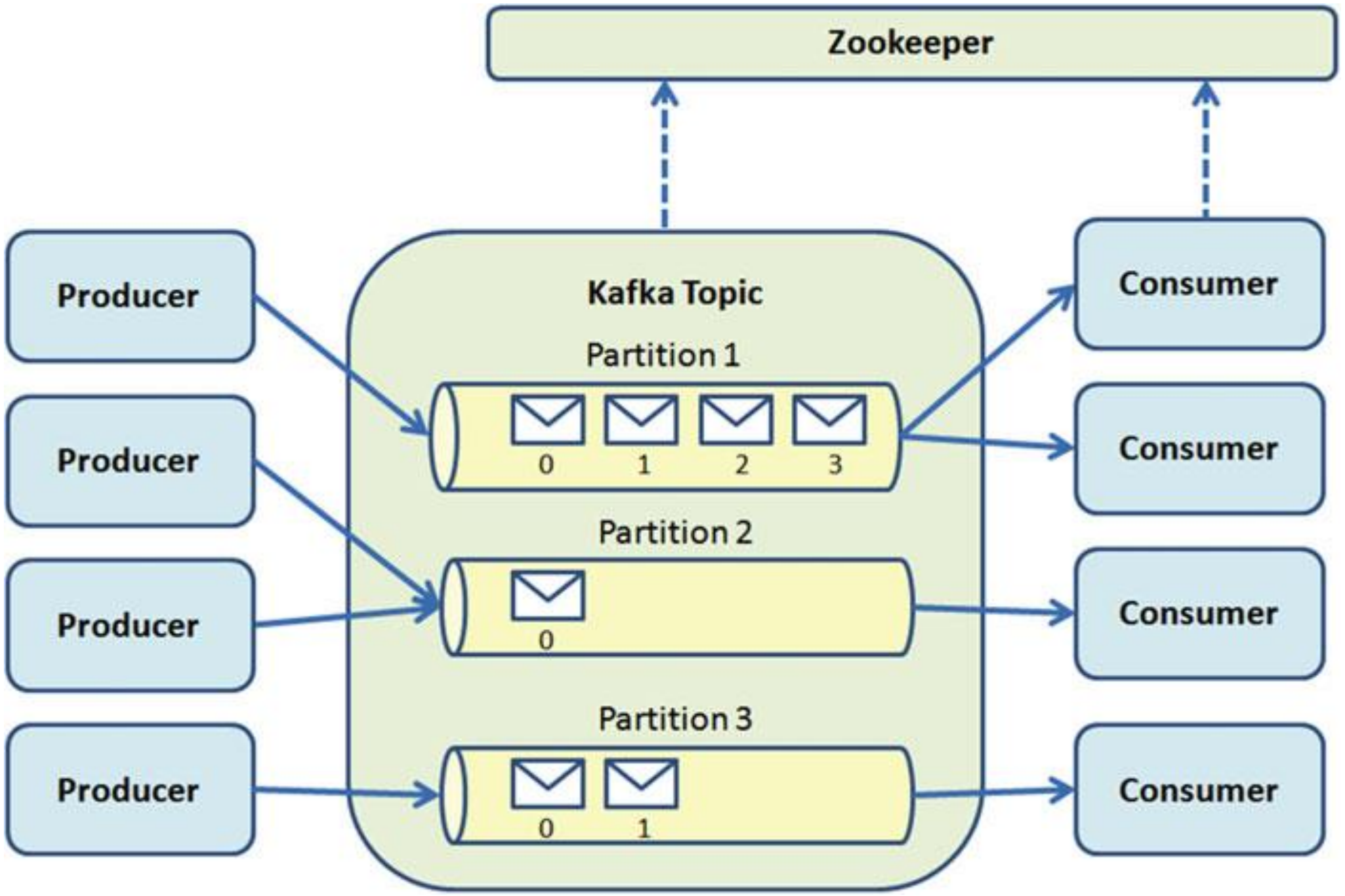
- Kafka goals:

  - **An API**. Supports custom implementation of producers and consumers.
  - **Low overhead**. Low network latency and low storage overhead with message persistence on disk.
  - **High throughput**. Publishes and subscribes millions of messages; supports data feeds and real time.
  - **Distributed**. Highly scalable architecture to handle low-latency delivery.
  - **High availability**. Auto balances consumers in case of failure.
  - **Fault tolerant**. Guarantees data integrity in case of failure.

Kafka Operation

- Kafka goals:
    - Messages are published to a Kafka topic, which is a message queue or a message category.
    - The Kafka topic runs in the Kafka broker, which is a server. Kafka brokers do not just run the topics, but also store the messages when required.
    - The consumers use the ZooKeeper service to get the information to track the messages (the data about a message state).

Kafka Operation



Kafka Partition

- **Segment files**. Internally, every partition is a logical log file, represented as a set of segment files with the same size. The partition is a sequence of ordered messages. When a message is published, the broker appends the message to the last segment of the file.
- **Offset**. The partitions are assigned to a unique sequential number called an offset. The offset is used to identify messages inside the partition. The partitions are replicated between the servers for fault tolerance.
- **Leaders**. Inside Kafka, each partition has one Kafka server as it leader. The other servers are followers. The leader of a partition coordinates the read and write requests for that partition. The followers asynchronously replicate the data from the leader.
- **Groups**. The consumers are organized into groups. Each consumer is represented as a process; one process belongs to only one group.

Kafka Log Compaction

- There are two types of retention: **finer-grained** (per message) and **coarser-grained** (time based). Log compaction is the process to pass from time-based to per-message retention.

- The retention policy can be set to per-topic (time based), size-based, and log compaction–based. Log compaction ensures the following:
  - Reads begin at offset 0; if the consumer begins at the start of the log, the messages are in the order that they were written.
  - Messages have sequential offsets that never change.
  - Message order is always preserved.
  - A group of background threads recopy log segment files; the records whose keys appear in the log head are removed.

Kafka Design

- **Storage**. The purpose of Kafka is to provide message processing. The main functions are caching and storing messages on a file system.

- **Retention** . If a message is consumed, the message is not wasted; it is retained, allowing message reconsumption.

- **Metadata**. In many messaging systems, the message metadata is kept at the server level. In Kafka, the message state is maintained at the consumer level. **This prevents: Multiple deliveries** of the same message; and **Losing messages** due to failures

Kafka Design

- **Online Transaction Processing (OLTP)**. Consumers store the state in ZooKeeper, but Kafka also allows the storage in OLTP external systems.
- **Push and pull**. Producers push the message to the broker and consumers pull the message from the broker.
- **Masterless**. Apache Kafka is masterless; you can remove any broker at any time. The metadata is maintained by ZooKeeper and shared with the customers.
- **Synchronous**. Producers have the option to be asynchronous or synchronous when sending messages to the broker.

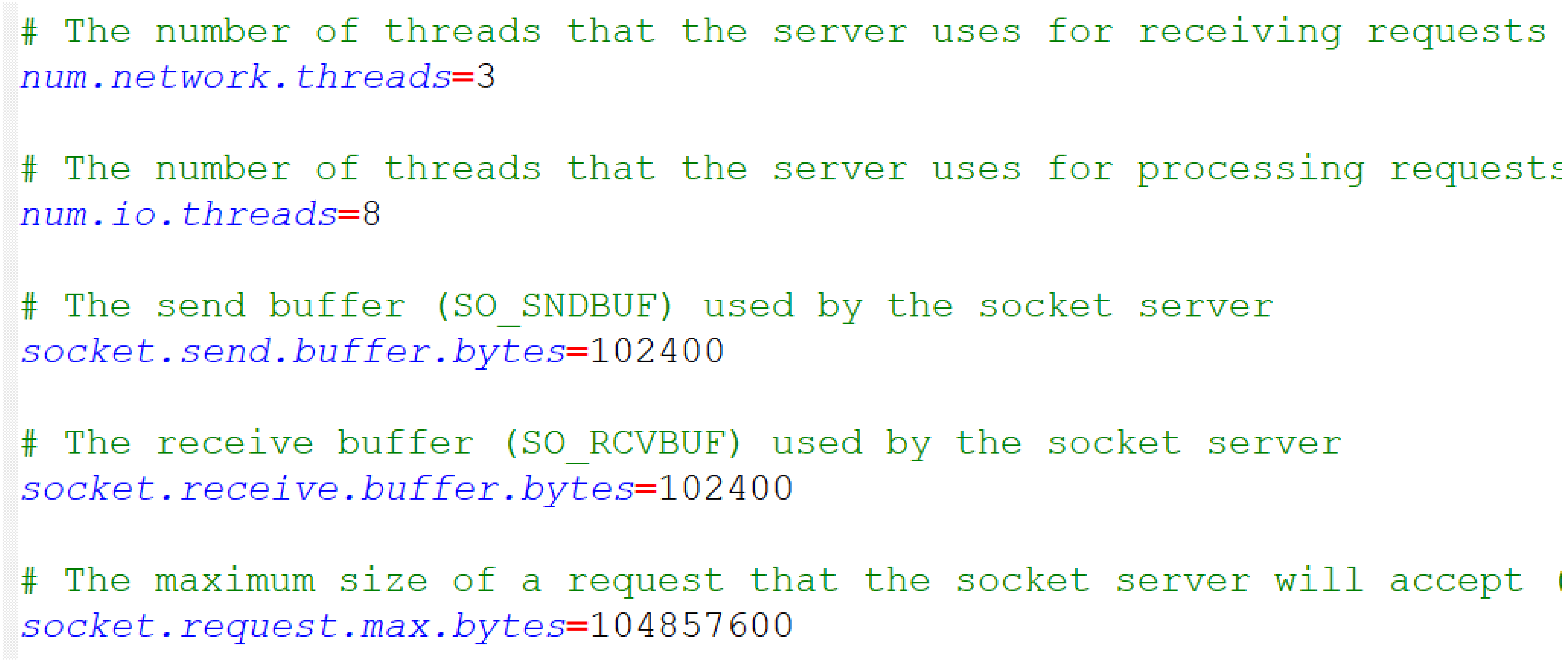
Kafka Message Compression

- There are cases where the network bandwidth is the bottleneck.
- There is a mechanism to compress groups of messages.
- When a group of messages is compressed, the network overhead is reduced.
- The lead broker is responsible for compressing messages, which lowers the network overhead but could also increase the load in the broker's CPU.
- Need to specify this configuration in the producer to use compression. ”compression.codec” or ”compressed.topics

Kafka Replication

- The **partitioning strategy** decision is made on the **broker side**.
- The **decision on how the message** is partitioned is made at the **producer end**.
- The broker stores the messages as they arrive. If you recall, the number of partitions configured for each topic is done in the Kafka broker.
- In replication, each partition has n replicas of a message (to handle n–1 failures). One replica acts as the leader.
- ZooKeeper knows who the replica leader is. The lead replica has a list of its follower replicas. The replicas store their part of the message in local logs.

Kafka configuration files



```
# The number of threads that the server uses for receiving requests
num.network.threads=3

# The number of threads that the server uses for processing requests
num.io.threads=8

# The send buffer (SO_SNDBUF) used by the socket server
socket.send.buffer.bytes=102400

# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=102400

# The maximum size of a request that the socket server will accept (
socket.request.max.bytes=104857600
```

Apache Kafka example Producer

Documentation Producer

```
from json import dumps
from time import sleep
from kafka import KafkaProducer

producer = KafkaProducer ( bootstrap_servers =['localhost:9092'],
value_serializer = lambda x : dumps ( x ).encode ( 'utf-8 '))

for e in range (1000) :
    data = {'number' : e }
    print(data)
    producer.send ('numtest' , value = data )
    sleep (5)
```

Apache Kafka example Consumer

Documentation Consumer

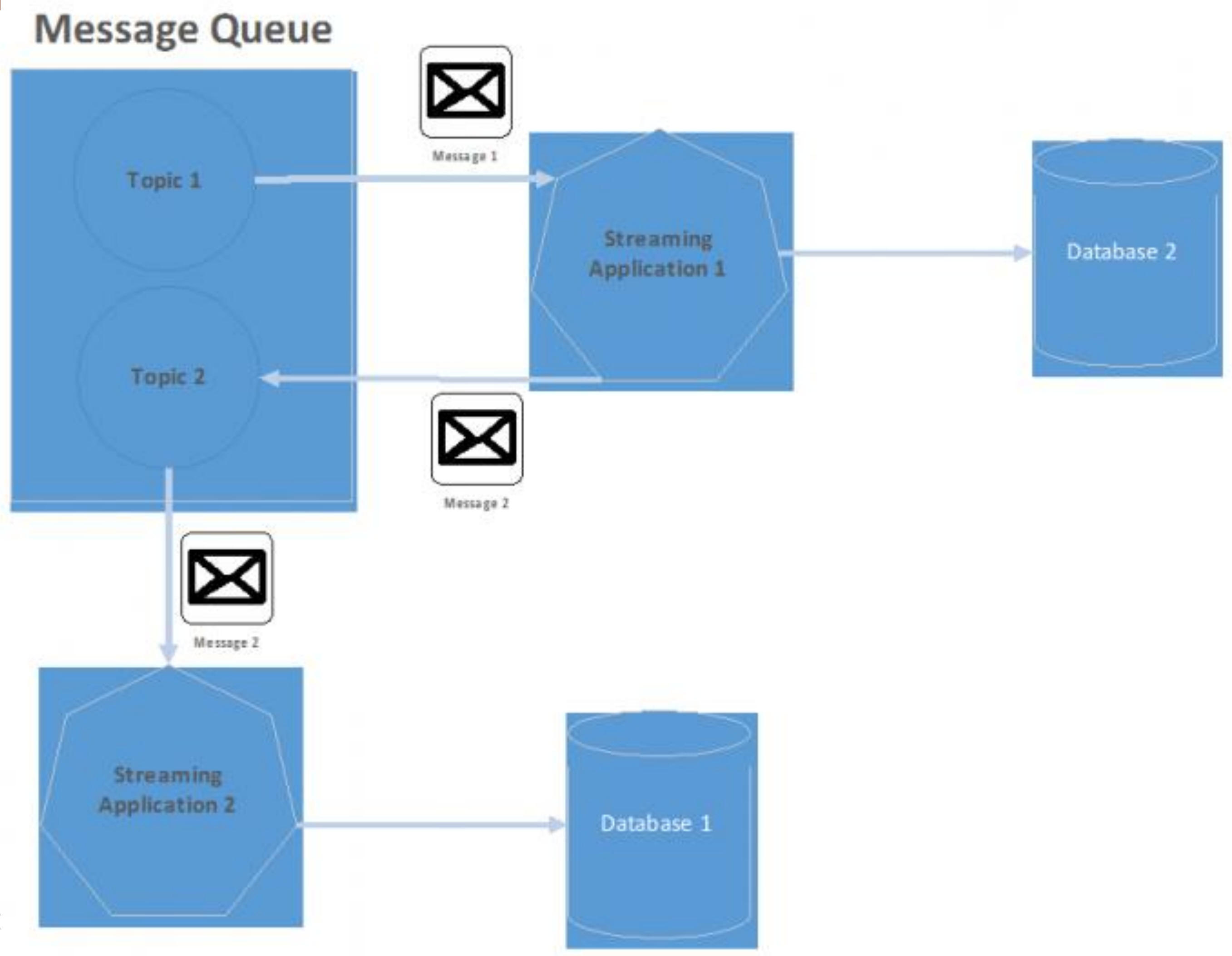
```
from json import loads
from kafka import KafkaConsumer

consumer = KafkaConsumer ( 'numtest' ,
bootstrap_servers=['localhost:9092'] , auto_offset_reset = 'earliest',
                          enable_auto_commit = True, group_id = 'my-
group',
                          value_deserializer = lambda x : loads (
x.decode ( 'utf-8') ) )
for message in consumer :
    message = message.value
    print ( message )
```

Using Kafka in Big Data Applications

- Kafka is becoming the standard tool for messaging in big data applications
- Not use databases as the one stop destination for all
- Earlier, due to lack of elegant storage systems, databases tend to be the only solution for any type of data store.
- If you use a database, over a period, the system will become highly complex to handle and expensive.

Using Kafka in Big Data Applications



Using Kafka in Big Data Applications

- Databases expects all data to be present in certain data formats. To fit all types of data in the expected data formats tends to make things more complex.

- We have improved the process of collecting data from different systems or devices.

- Each of those systems has different data formats and data types. The same data is also utilized to feed in different data pipelines such as real-time alerting, batch reports, etc.

- Kafka is apt for these situations because of the following features:

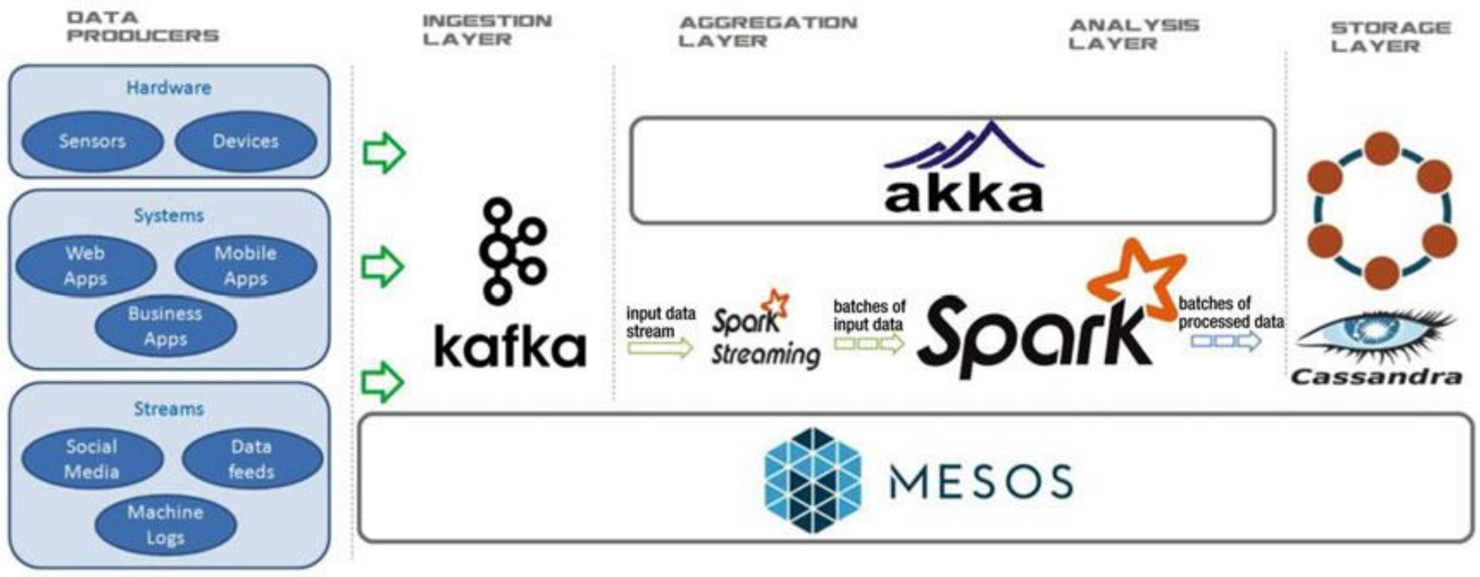
Using Kafka in Big Data Applications

- It has the support to store data of **any types and formats**
- It uses commodity hardware for **storing high volumes of data**
- It is a **high-performance and scalable system**
- It stores data on disk and can be used to serve different data pipelines; it can be used in **real-time event processing** and **batch processing**
- Due to data and system redundancy, it is **highly reliable**, which is an important requirement in enterprise grade production-deployed big data application

Managing high volumes in Kafka

- We need to choose the right set of hardware and perform appropriate capacity planning

- We need to think of following aspects: (i) High volume of writes or high message writing throughput; (ii) High volumes of reads or high message reading throughput; (iii) High volume of replication rate; and (iv) High disk flush or I/O



Managing high volumes in Kafka

- In the case of a high volume of writes, producers should have more capacity to buffer records.
- Since batching is always suggested for a high volume of writes, you would require more **network bandwidth for a connection between the producer component and the broker.**
- Similar is the case with high volume reads where you would need more **memory for consumer application.**
- For brokers, you need to give more thought to the hardware as with high volumes, brokers do majority of the work. Brokers are multi-threaded applications.

Managing high volumes in Kafka

- Some of the techniques that you can use while writing or reading data:
- **Message compression**: The producer generates the compression type of all the data. The value for the compression type property are none, GZIP, Snappy, or lZ4.
- **Message batches**: This property is specific to producers in the asynchronous mode. A small-sized batch may reduce throughput, and setting batch size to zero will disable the batch size.

Managing high volumes in Kafka

- **Asynchronous send**: It offers the capability to do sends on a **separate thread** that isolates the network I/O from the thread doing computation and allows multiple messages to be sent in a single batch.

- **Linger time**: The producer sends buffer once it is available and does not wait for any other trigger. Linger time allows us to set the maximum time in which data can be buffered before the producer sends it for storage.

- **Fetch size**: The number of partitions defines the maximum number of consumers from the same consumer group who can read messages from it.

Kafka message delivery semantics

- It is important to understand that the way messages are delivered by the producer effects the manner in which the consumer would receive the messages.
- This would ultimately have impact on applications processing those consumer received messages.
- There are three types of message delivery semantics:
  - **At most once**: messages are only read or written once (fire and forget).
  - **At least once**: messages are read or written at least once, and they are never lost.
  - **Exactly Once**: This is the most favorable delivery semantics as it ensures messages are delivered once and only once.

Big data and Kafka common usage patterns

- One of the common usage patterns of Kafka is to use it as a streaming data platform.

- It supports storing streaming data from varied sources, and that data can later be processed in real time or in batch.

Big data and Kafka common usage patterns

- Kafka is used in micro-services or IOT-based architecture.
- These kinds of architecture are driven by request responses and event-based approaches with Kafka as a central piece of it.
- Services or IOT devices raise events that are received by Kafka brokers. The messages can then be used for further processing.
- Overall, Kafka, due to its high scalability and performance-driven design, is used as an event store for many different types of applications, including big data applications.

Kafka and data governance

- In any enterprise grade Kafka deployment, you need to build a **solid governance** framework to **ensure security of confidential** data along with who is dealing with data and what kind of operations are performed on data.
- Moreover, governance framework ensures who can access what data and who can perform operations on data elements.
- There are tools available such as Apache Atlas and Apache Ranger, which will help you define a proper governance framework around Kafka.
- The **fundamental data element in Kafka is Topic**. You should define all your governance processes around Topic data element.

Alerting and monitoring

- If you have Kafka as a centralized messaging system in your data pipeline and many applications are dependent on it, any cluster disaster or bottleneck in the Kafka cluster may affect the performance of all application dependent on Kafka.

- Hence, it is important to have a proper **alerting and monitor system** in place that gives us **important information about the health of the Kafka cluster.**

- **Avoid data loss**: Sometimes it may happen that **topic partitions are under replicated**, meaning they have fewer number of replicas available in the cluster.

Alerting and monitoring

- **Producer performance**: The alerting and monitoring system will also help us improve the producer performance by observing its metrics.

- **Consumer performance**: We may also observe that the consumer is not able to process data as fast as the producer is producing it, or that the consumer is not able to consume data due to some network bandwidth issue.

- **Data availability**: Sometimes, the leaders for partitions are not assigned, or it takes time for the assignment to happen

Apache Kafka example Producer

```
from json import dumps
from time import sleep
from kafka import KafkaProducer

producer = KafkaProducer ( bootstrap_servers =['localhost:9092'],
value_serializer = lambda x : dumps ( x ).encode ( 'utf-8 '))

for e in range (1000) :
    data = {'number' : e }
    print(data)
    producer.send ('numtest' , value = data )
    sleep (5)
```

Apache Kafka example Consumer

```
from json import loads
from kafka import KafkaConsumer

consumer = KafkaConsumer ( 'numtest' ,
bootstrap_servers=['localhost:9092'] , auto_offset_reset = 'earliest',
                          enable_auto_commit = True, group_id = 'my-
group',
                          value_deserializer = lambda x : loads (
x.decode ( 'utf-8') ) )
for message in consumer :
    message = message.value
    print ( message )
```

Exercises

1. Create two producers that send in parallel random numbers with different time intervals. Design a consumer that receives these values.
2. Create N producers that send the information of each attribute of the Iris dataset. The consumer will merge all the data at the end.
3. Create N producers that send 100 random words. The M consumers will count the number of words that contains less or greater than 3 vowels.

Threads in Python

```
def producer_one(topic):
        ….
def producer_two(topic):
        ….
```

```

# Start the two producers in parallel
topic = 'random_numbers'
t1 = threading.Thread(target=producer_one, args=(topic,))
t2 = threading.Thread(target=producer_two, args=(topic,))
t1.start()
t2.start()
```

Work to be done to the next class

- Replicate the Producers and Consumers developed in the previous exercises for your dataset

- Start the paper
  - Abstract
  - Introduction
  - Related-work

- Use template in Moodle for the paper. You can use word or latex

References

- Estrada, R., & Ruiz, I. (2016). Big data smack. Apress, Berkeley, CA.
- Kumar, M., & Singh, C. (2017). Building Data Streaming Applications with Apache Kafka. Packt Publishing Ltd

UNIVERSIDADE
PORTUCALENSE

Do conhecimento à prática.