Messaging Systems: Introduction

This lesson gives a brief overview of messaging systems.

We'll cover the following

Goal#

Design a distributed messaging system that can reliably transfer a high throughput of messages between different entities.

Background#

One of the common challenges among distributed systems is handling a continuous influx of data from multiple sources. Imagine a log aggregation service that is receiving hundreds of log entries per second from different sources. The function of this log aggregation service is to store these logs on disk at a shared server and also build an index so that the logs can be searched later. A few challenges of this service are:

1. How will the log aggregation service handle a spike of messages? If the service can handle (or buffer) 500 messages per second, what will

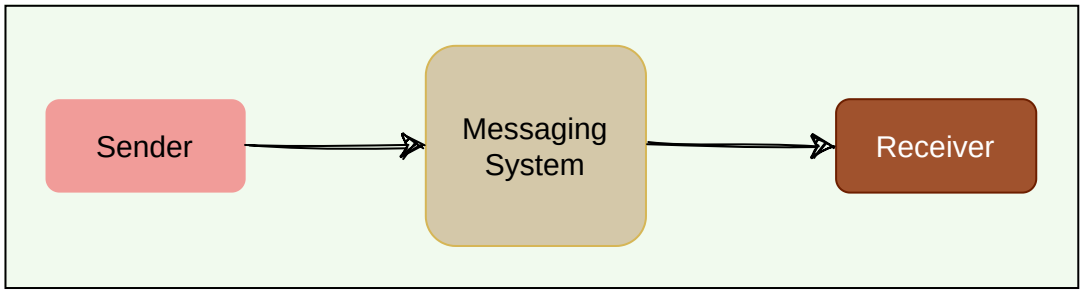
happen if it starts receiving a higher number of messages per second? If we decide to have multiple instances of the log aggregation service, how do we divide the work among these instances?

2. How can we receive messages from different types of sources? The sources producing (or consuming) these logs need to decide upon a common protocol and data format to send log messages to the log aggregation service. This leads us to a strongly coupled architecture between the producer and consumer of the log messages.
3. What will happen to the log messages if the log aggregation service is down or unresponsive for some time?

To efficiently manage such scenarios, distributed systems depend upon a messaging system.

What is a messaging system?

A **messaging system** is responsible for transferring data among services, applications, processes, or servers. Such a system helps **decouple** different parts of a distributed system by providing an **asynchronous** way of transferring messaging between the sender and the receiver. Hence, all senders (or producers) and receivers (or consumers) focus on the data/message without worrying about the mechanism used to share the data.

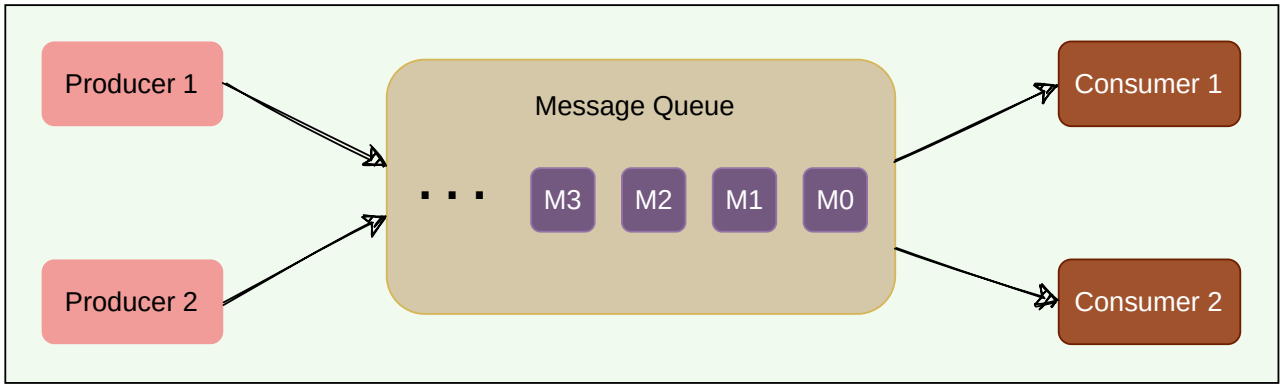


Messaging system

There are two common ways to handle messages: **Queuing** and **Publish-Subscribe**.

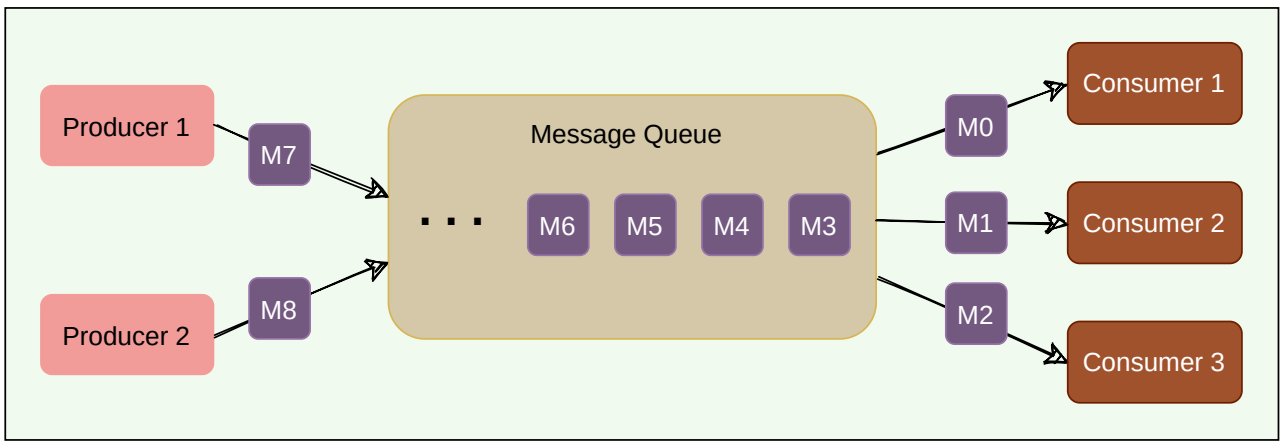
Queue#

In the queuing model, messages are stored sequentially in a queue. Producers push messages to the rear of the queue, and consumers extract the messages from the front of the queue.



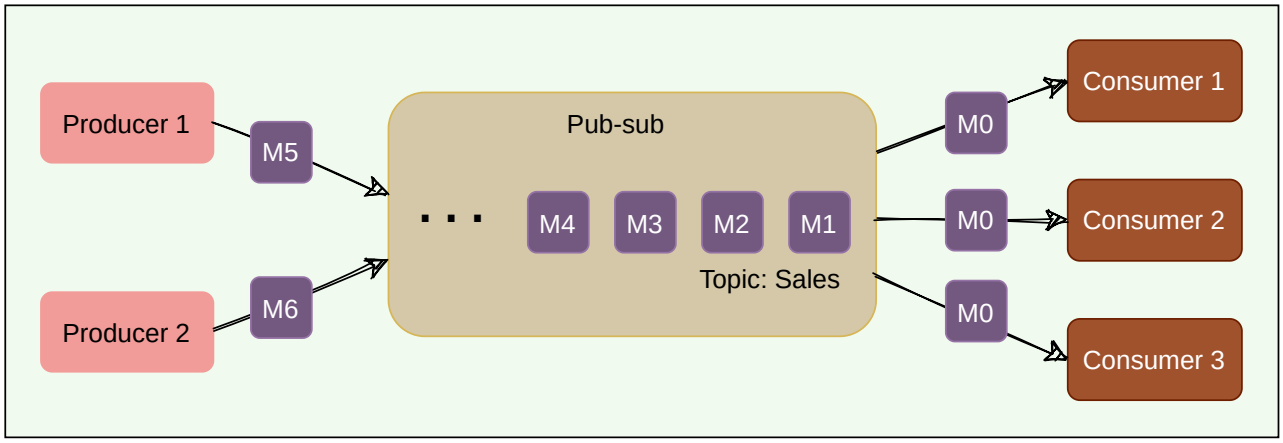
Message queue

A particular message can be consumed by a maximum of one consumer only. Once a consumer grabs a message, it is removed from the queue such that the next consumer will get the next message. This is a great model for distributing message-processing among multiple consumers. But this also limits the system as multiple consumers cannot read the same message from the queue.



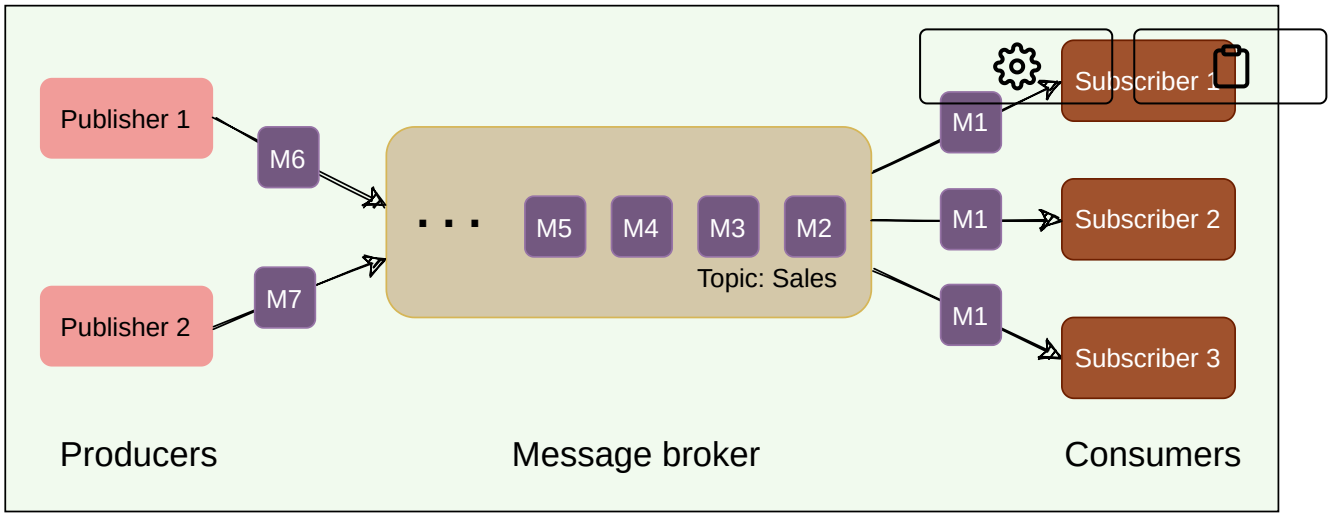
Publish-subscribe messaging system#

In the pub-sub (short for publish-subscribe) model, messages are divided into topics. A publisher (or a producer) sends a message to a topic that gets stored in the messaging system under that topic. Subscribers (or the consumer) subscribe to a topic to receive every message published to that topic. Unlike the Queuing model, the pub-sub model allows multiple consumers to get the same message; if two consumers subscribe to the same topic, they will receive all messages published to that topic.



Pub-sub messaging system

The messaging system that stores and maintains the messages is commonly known as the message **broker**. It provides a loose coupling between publishers and subscribers, or producers and consumers of data.



Message broker

The message broker stores published messages in a queue, and subscribers read them from the queue. Hence, subscribers and publishers do not have to be synchronized. This **loose coupling** enables subscribers and publishers to read and write messages at different rates.

The messaging system's ability to store messages provides **fault-tolerance**, so messages do not get lost between the time they are produced and the time they are consumed.

To summarize, a message system is deployed in an application stack for the following reasons:

1. **Messaging buffering**. To provide a buffering mechanism in front of processing (i.e., to deal with temporary incoming message spikes that are greater than what the processing app can deal with). This enables the system to safely deal with spikes in workloads by temporarily storing data until it is ready for processing.
2. **Guarantee of message delivery**. Allows producers to publish messages with assurance that the message will eventually be delivered if the consuming application is unable to receive the message when it is published.
3. **Providing abstraction**. A messaging system provides an architectural separation between the consumers of messages and the applications producing the messages.

4. **Enabling scale**. Provides a flexible and highly configurable architecture that enables many producers to deliver messages to multiple consumers.

← Back

Next →

Quiz: Cassandra

Kafka: Introduction

Mark as Completed

Report an Issue