Introduction to Pub-sub

Learn about the use cases of the pub-sub system, how to define its requirements, and design the API for it.

We'll cover the following

Pub-sub messaging offers asynchronous communication. Let’s explore the use cases where it is beneficial to have a pub-sub system.

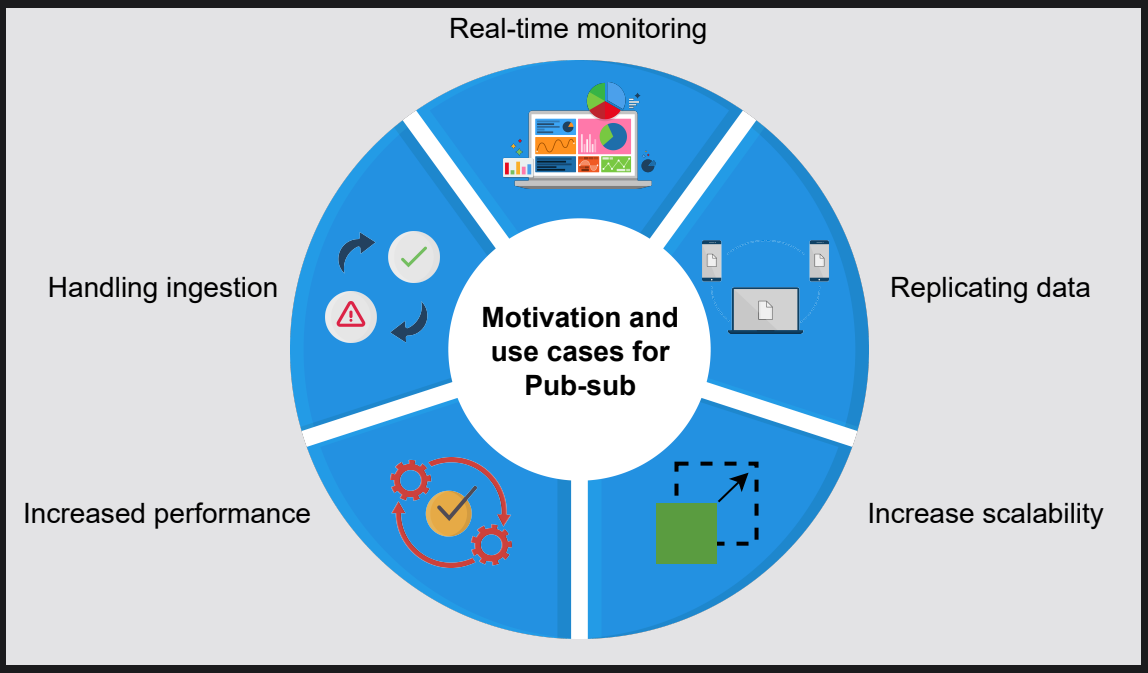
Use cases of pub-sub

A few use cases of pub-sub are listed below:

- **Improved performance**: The pub-sub system enables push-based distribution, alleviating the need for message recipients to check for new information and changes regularly. It encourages faster response times and lowers the delivery latency.
- **Handling ingestion**: The pub-sub helps in handling log ingestion. The user-interaction data can help us figure out useful analyses about the behavior of users. We can ingest a large amount of data to the pub-sub system, so much so that it can deliver the data to any analytical system t understand the behavior patterns of users. Moreover, we can also log the

details of the event that's happening while completing a request from the user. Large services like Meta use a pub-sub system called Scribe to know exactly who needs what data, and remove or archive processed or unwanted data. Doing this is necessary to manage an enormous amount of data.

- **Real-time monitoring**: Raw or processed messages of an application or system can be provided to multiple applications to monitor a system in real time.
- **Replicating data**: The pub-sub system can be used to distribute changes. For example, in a leader-follower protocol, the leader sends the changes to its followers via a pub-sub system. It allows followers to update their data asynchronously. The distributed caches can also refresh themselves by receiving the modifications asynchronously. Along the same lines, applications like WhatsApp that allow multiple views of the same conversation—for example, on a mobile phone and a computer's browser—can elegantly work using a pub-sub, where multiple views can act either as a publisher or a subscriber.



Motivation and use cases of the pub-sub system

Points to Ponder

Question 1

What are the similarities and differences between a pub-sub system and queues?

Show Answer

1 of 2

Requirements

We aim to design a pub-sub system that has the following requirements.

Functional requirements

Let's specify the functional requirements of a pub-sub system:

- **Create a topic**: The producer should be able to create a topic.
- **Write messages**: Producers should be able to write messages to the topic.
- **Subscription**: Consumers should be able to subscribe to the topic to receive messages.
- **Read messages**: The consumer should be able to read messages from the topic.
- **Specify retention time**: The consumers should be able to specify the retention time after which the message should be deleted from the system.
- **Delete messages**: A message should be deleted from the topic or system after a certain retention period as defined by the user of the system.

You're designing a real-time chat application that has channels for different discussions. Explain how you would use a pub-sub system for its messaging component.

Describe the mapping of the chatting application onto the following pub-sub components:

- Topics
- Subscribers
- How message delivery would work

Write your answer in the widget below.

Want to know the correct answer?

How you would use a pub-sub system in a chat application?

H1 H2 H3 | B I | | | |

Provide your answer here!

Non-functional requirements

We consider the following non-functional requirements when designing a pub-sub system:

- **Scalable**: The system should scale with an increasing number of topics and increasing writing (by producers) and reading (by consumers) load.
- **Available**: The system should be highly available, so that producers can add their data and consumers can read data from it anytime.
- **Durability**: The system should be durable. Messages accepted from producers must not be lost and should be delivered to the intended

subscribers.

- **Fault tolerance**: Our system should be able to operate in the event of failures.
- **Concurrent**: The system should handle concurrency issues where reading and writing are performed simultaneously.

API Design

We'll exclude some parameters from the functions below, such as the producer or consumer's identifier. Let's assume that this information is available from the underlying connection context. The API design for this problem is as follows:

Create a topic

The API call to create a topic should look like this:

```
create(topic_ID, topic_name)
```

This function returns an acknowledgment if it successfully creates a topic, or an error if it fails to do so.

Parameter	Description
`topic_ID`	It uniquely identifies the topic.
`topic_name`	It contains the name of the topic.

Write a message

The API call to write into the pub-sub system should look like this:

```
write(topic_ID, message)
```

The API call will write a `message` into a topic with an ID of `topic_ID`. Each message can have a maximum size of 1 MB. This function will return an

acknowledgment if it successfully places the data in the systems, or an appropriate error if it fails.

Parameter	Description
`message`	The message to be written in the system.

Read a message

The API call to read data from the system should look like this:

```
read(topic_ID)
```

The topic is found using `topic_ID`, and the call will return an object containing the message to the caller.

Parameter	Description
`topic_ID`	It is the ID of the topic against which the message will be read.

Subscribe to a topic

The API call to subscribe to a topic from the system should look like this:

```
subscribe(topic_ID)
```

The function adds the consumer as a subscriber to the topic that has the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic to which the consumer will be subscribed.

Unsubscribe from a topic

The API call to unsubscribe from a topic from the system should look like this:

```
unsubscribe(topic_ID)
```

The function removes the consumer as a subscriber from the topic that has the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic against which the consumers will be unsubscribed.

Delete a topic

The API call to delete a topic from the system should look like this:

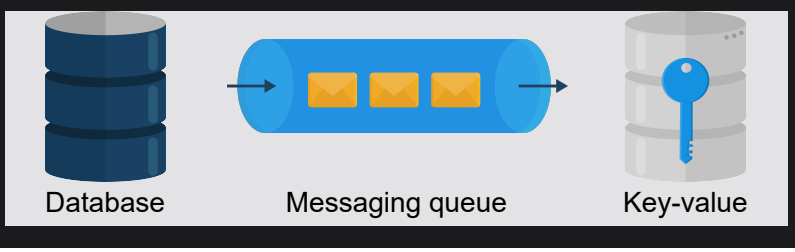
```
delete_topic(topic_ID)
```

The function deletes the topic on the basis of the `topic_ID`.

Parameter	Description
`topic_ID`	The ID of the topic which is to be deleted.

Building blocks we will use

The design of pub-sub utilizes many building blocks that have been discussed in the initial chapters. We'll consider the following lessons on building blocks.



- **Database**: We'll use databases to store information like subscription details.
- **Distributed messaging queue**: We'll use use a messaging queue to store messages sent by the producer.
- **Key-value**: We'll use a key-value store to hold information about consumers.

In the next lesson, we'll focus on designing a pub-sub system.

Back

Completed

Next