Join

Back To Course Home

Grokking Modern System Design Interview for Engineers & Managers

0% completed

Search Course

Design of a Pub-sub System

Dive into designing a pub-sub system and its components.

We'll cover the following ☐

First design#

In the previous lesson, we discussed that a producer writes into topics, and consumers subscribe to a topic to read messages from that topic. Since new messages are added at the end of the queue, we can use distributed messaging queues for topics.

The components we'll need have been listed below:

- **Topic queue**: Each topic will be a distributed messaging queue so we can store the messages sent to us from the producer. A producer will write their messages to that queue.
- **Database**: We'll use a relational database that will store the subscription details. For example, we need to store which consumer has subscribed to which topic so we can provide the consumers with their desired messages. We'll use a relational database since our consumer-related data is structured and we want to ensure our data integrity.
- **Message director**: This service will read the message from the topic queue, fetch the consumers from the database, and send the message to the consumer queue.
- **Consumer queue**: The message from the topic queue will be copied to the consumer's queue so the consumer can read the message. For each consumer, we'll

define a separate distributed queue.

- **Subscriber**: When the consumer requests a subscription to a topic, this service will add an entry into the database.

The consumer will subscribe to a topic, and the system will add the subscriber's details to the database. The producer will write into the topics, and the message director will read the message from the queue, fetch the details to whom it should add the message, and send it to them. The consumers will consume the message from their queue.

> **Note**: We'll use fail-over services for the message director and subscriber to guard against failures.

Using the distributed messaging queue

Using the distributed messaging queues makes our design simple. However, the huge number of queues needed is a significant concern. If we have millions of subscribers for thousands of topics, defining and maintaining millions of queues is expensive. Moreover, we'll copy the same message for a topic in all subscriber queues, which is unnecessary duplication and takes up space.

Points to Ponder

Question 1

Is there a way to avoid maintaining a separate queue for each reader?

Show Answer

1 of 2

Second design#

Let’s consider another approach to designing a pub-sub system.

High-level design#

At a high level, the pub-sub system will have the following components:

- **Broker**: This server will handle the messages. It will store the messages sent from the producer and allow the consumers to read them.
- **Cluster manager**: We’ll have numerous broker servers to cater to our scalability needs. We need a cluster manager to supervise the broker’s health. It will notify us if a broker fails.
- **Storage**: We’ll use a relational database to store consumer details, such as subscription information and retention period.
- **Consumer manager**: This is responsible for managing the consumers. For example, it will verify if the consumer is authorized to read a message from a certain topic or not.

Besides these components, we also have the following design considerations:

- **Acknowledgment**: An acknowledgment is used to notify the producer that the received message has been stored successfully. The system will wait for an acknowledgment from the consumer if it has successfully consumed the message.

- **Retention time**: The consumers can specify the retention period time of their messages. The default will be seven days, but it is configurable. Some applications like banking applications require the data to be stored for a few weeks as a business requirement, while an analytical application might not need the data after consumption.

High-level design of a pub-sub system

Let’s understand the role of each component in detail.

Broker#

The broker server is the core component of our pub-sub system. It will handle write and read requests. A broker will have multiple topics where each topic can have multiple **partitions** associated with it. We use partitions to store messages in the local storage for persistence. Consequently, this improves availability. Partitions contain messages encapsulated in **segments**. Segments help identify the start and end of a message using an offset address. Using segments, consumers consume the message of their choice from a partition by reading from a specific offset address. The illustration below depicts the concept that has been described above.

A depiction of how messages are stored within segments inside a partition

As we know, a **topic** is a persistent sequence of messages stored in the local storage of the broker. Once the data has been added to the topic, it cannot be modified. Reading and writing a message from or to a topic is an I/O task for computers, and scaling such tasks is challenging. This is the reason we split the topics into multiple partitions. The data belonging to a single topic can be present in numerous partitions. For example, let's assume have Topic A and we allocate three partitions for it. The producers will send their messages to the relevant topic. The messages received will be sent to various partitions on basis of the round-robin algorithm. We'll use a variation of round-robin: weighted round-robin. The following slides show how the messages are stored in various partitions belonging to a single topic.

Producers create messages of the same topic and send them to the system

1 of 9

Points to Ponder

Question 1

Strict ordering ensures that the messages are stored in the order in which they are produced. How can we ensure strict ordering for our messages?

Show Answer

1 of 4

We’ll allocate the partitions to various brokers in the system. This just means that different partitions of the same topic will be in different brokers. We’ll follow strict ordering in partitions by adding newer content at the end of existing messages.

Consider the slides below. We have various brokers in our system. Each broker has different topics. The topic is divided into multiple partitions.

A broker contains multiple topics

1 of 3

We discussed that a message will be stored in a segment. We’ll identify each segment using an offset. Since these are immutable records, the readers are independent and they can read messages anywhere from this file using the necessary API functions. The following slides show the segment level detail.

A new entry will be added at the end of the file

1 of 2

The broker solved the problems that our first design had. We avoided a large number of queues by partitioning the topic. We introduced parallelism using partitions that avoided bottlenecks while consuming the message.

Cluster manager#

We’ll have multiple brokers in our cluster. The cluster manager will perform the following tasks:

- **Broker and topics registry**: This stores the list of topics for each broker.
- **Manage replication**: The cluster manager manages replication by using the leader-follower approach. One of the brokers is the leader. If it fails, the manager decides who the next leader is. In case the follower fails, it adds a new broker and makes sure to turn it into an updated follower. It updates the metadata accordingly. We’ll keep three replicas of each partition on different brokers.

Replication at the partitioning level

Consumer manager#

The consumer manager will manage the consumers. It has the following responsibilities:

- **Verify the consumer**: The manager will fetch the data from the database and verify if the consumer is allowed to read a certain message. For example, if the consumer has subscribed to Topic A (but not to Topic B), then it should not be allowed to read from Topic B. The consumer manager verifies the consumer’s request.

- **Retention time management**: The manager will also verify if the consumer is allowed to read the specific message or not. If, according to its retention time, the message should be inaccessible to the consumer, then it will not allow the consumer to read the message.

- **Message receiving options management**: There are two methods for consumers to get data. The first is that our system pushes the data to its consumers. This method may result in overloading the consumers with continuous messages. Another approach is for consumers to request the system to read data from a specific topic. The drawback is that a few consumers might want to know about a message as soon as it is published, but we do not support this function.

  Therefore, we'll support both techniques. Each consumer will inform the broker that it wants the data to be pushed automatically or it needs the data to read itself. We can avoid overloading the consumer and also provide liberty to the consumer. We'll save this information in the relational database along with other consumer details.

- **Allow multiple reads**: The consumer manager stores the offset information of each consumer. We'll use a key-value to store offset information against each consumer. It allows fast fetching and increases the availability of the consumers. If Consumer 1 has read from offset 0 and has sent the acknowledgment, we'll store it. So, when the consumer wants to read again, we can provide the next offset to the reader for reading the message.

Finalized design#

The finalized design of our pub-sub system is shown below.

Finalized design

Conclusion#

We saw two designs of pub-sub, using queues and using our custom storage optimized for writing and reading small-sized data.

There are numerous use cases of a pub-sub. Due to decoupling between producers and consumers, the system can scale dynamically, and the failures are well-contained. Additionally, due to proper accounting of data consumption, the pub-sub is a system of choice for a large-scale system that produces enormous data. We can determine precisely which data is needed and not needed.

Introduction to Pub-sub

Next

System Design: The Rate Limiter

Mark as Completed

Report an Issue