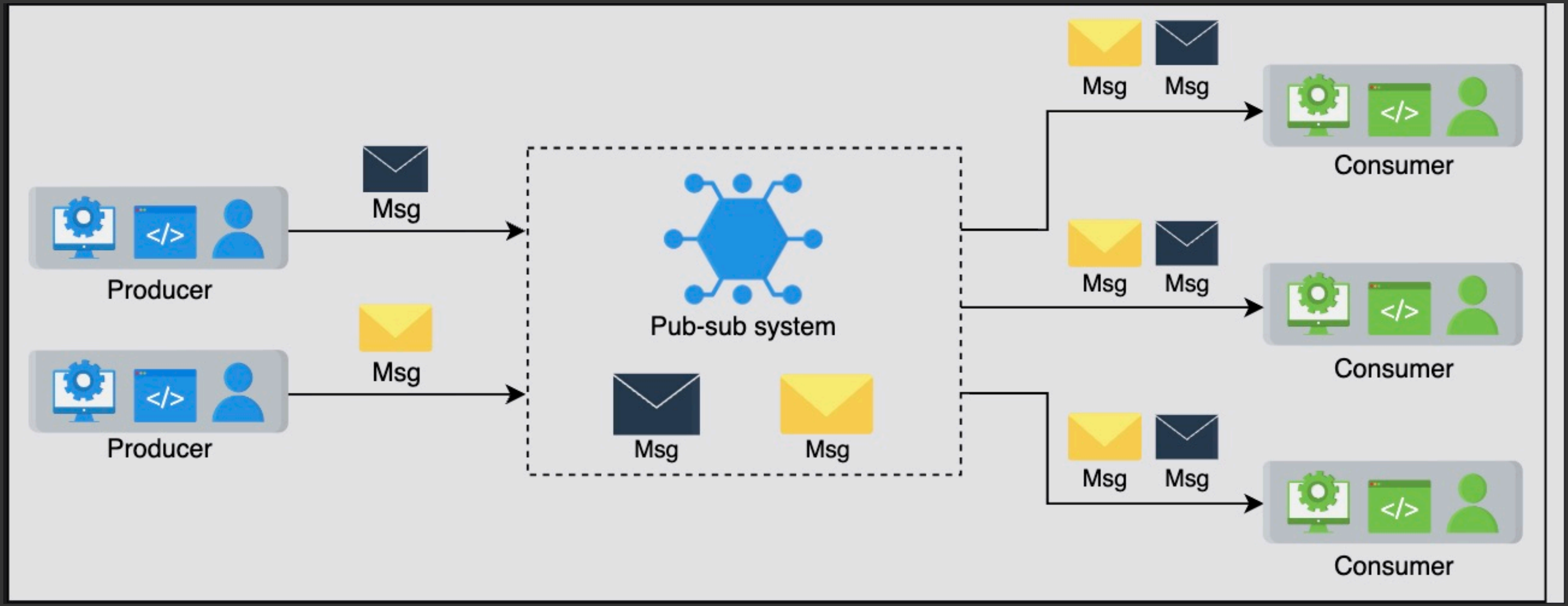
what is Pub - Sub ?

Publish-subscribe messaging, often known as pub-sub messaging, is an asynchronous service-to-service communication method that's popular in serverless and microservices architectures. Messages can be sent asynchronously to different subsystems of a system using the pub-sub system. All the services subscribed to the pub-sub model receive the message that's pushed into the system.



Introduction to Pub-Sub

Use cases of Pub-Sub

- Improved performance
- Handling ingestion
- Real-time monitoring
- Replicating Data.

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

Hide Answer
The pub-sub system and queues are similar because they deliver information that's produced by the producer to the consumer. The difference is that only one consumer consumes a message in the queue, while there can be multiple consumers of the same message in a pub-sub system.

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.

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.

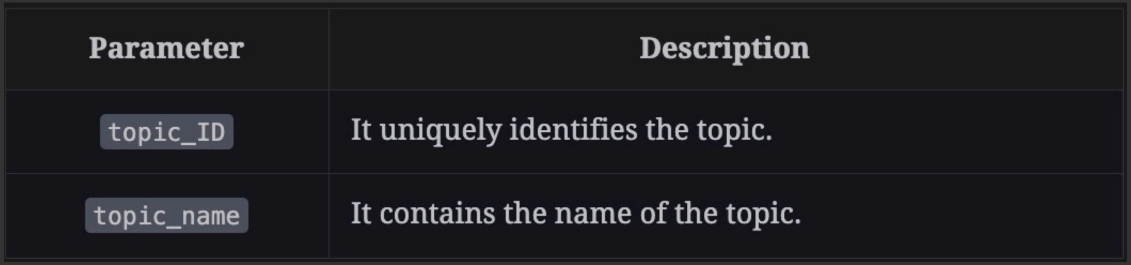
Fault tolerance: Our system should be able to operate in the event of failures.

Concurrent: The system should handle concurrency issues where reading and writing are performed simultaneously.

Create a topic

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

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



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)
```

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)
```

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)
```

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)
```

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)
```

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

The components we'll need have been listed below:

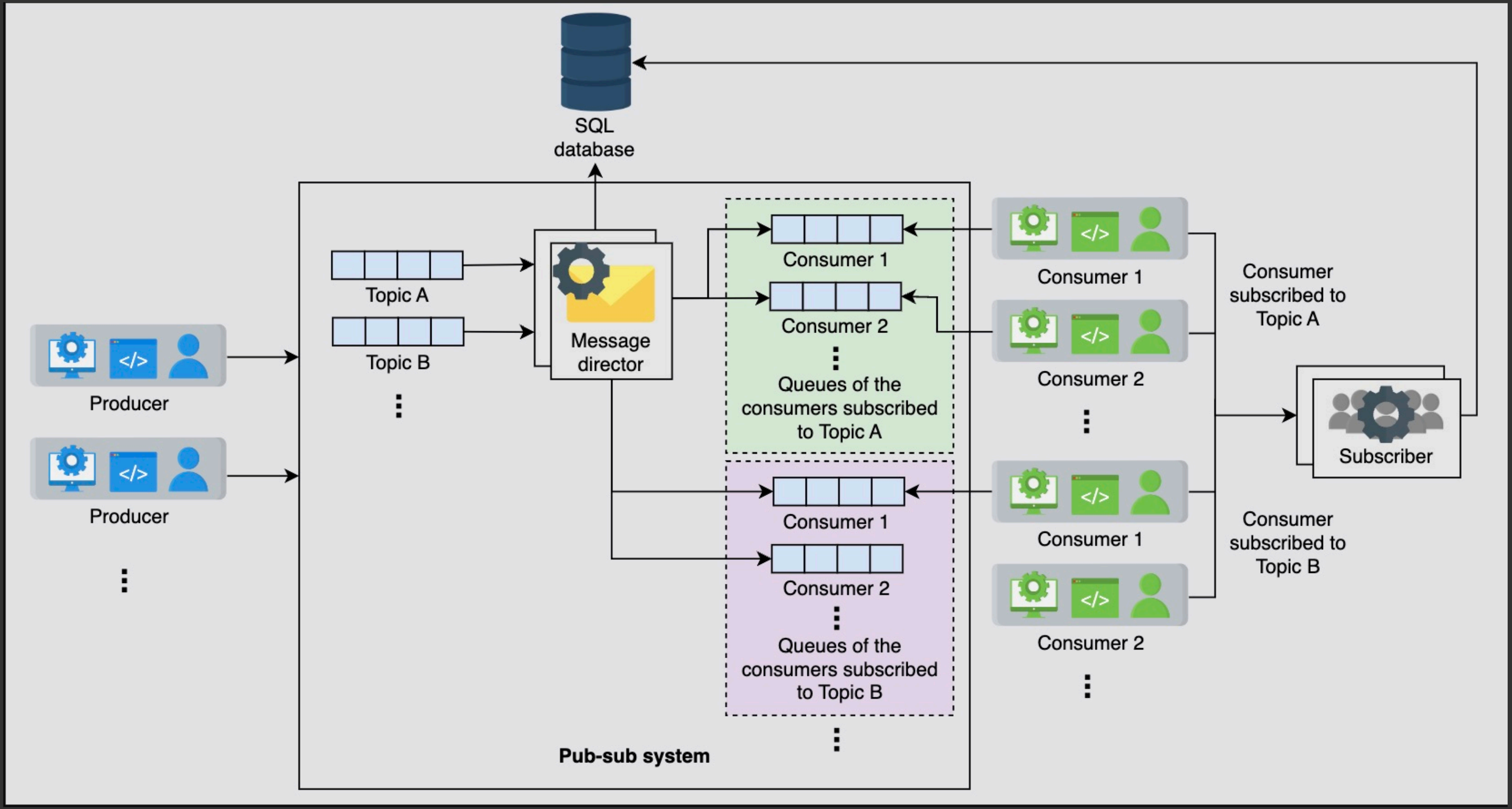
Topic queue: Each topic will be a distributed messaging queue so we can store the messages sent to us from the producer. A producer will write their messages to that queue.

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.



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

Hide Answer
In messaging queues, the message disappears after the reader consumes it. So, what if we add a counter for each message? The counter value decrements as a subscriber consumes the message. It does not delete the message until the counter becomes zero. Now, we don’t need to keep a separate queue for each reader.

What is the problem with the previous approach?

Hide Answer
The unread messages can become a bottleneck if we use the conventional queue API. For example, if 9 out of 10 readers have consumed the message present at the start of the queue, then that message won’t be deleted until the tenth consumer has also consumed the message, and the first nine consumers won’t be able to move forward.

We’ll need to change the storage interface so that consumers can independently consume data. Our system will need to keep sufficient metadata and track what information each consumer has consumed and delete a message when the information has been consumed by all consumers. It resembles with reference count mechanism in Linux’s hard link of files.

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.

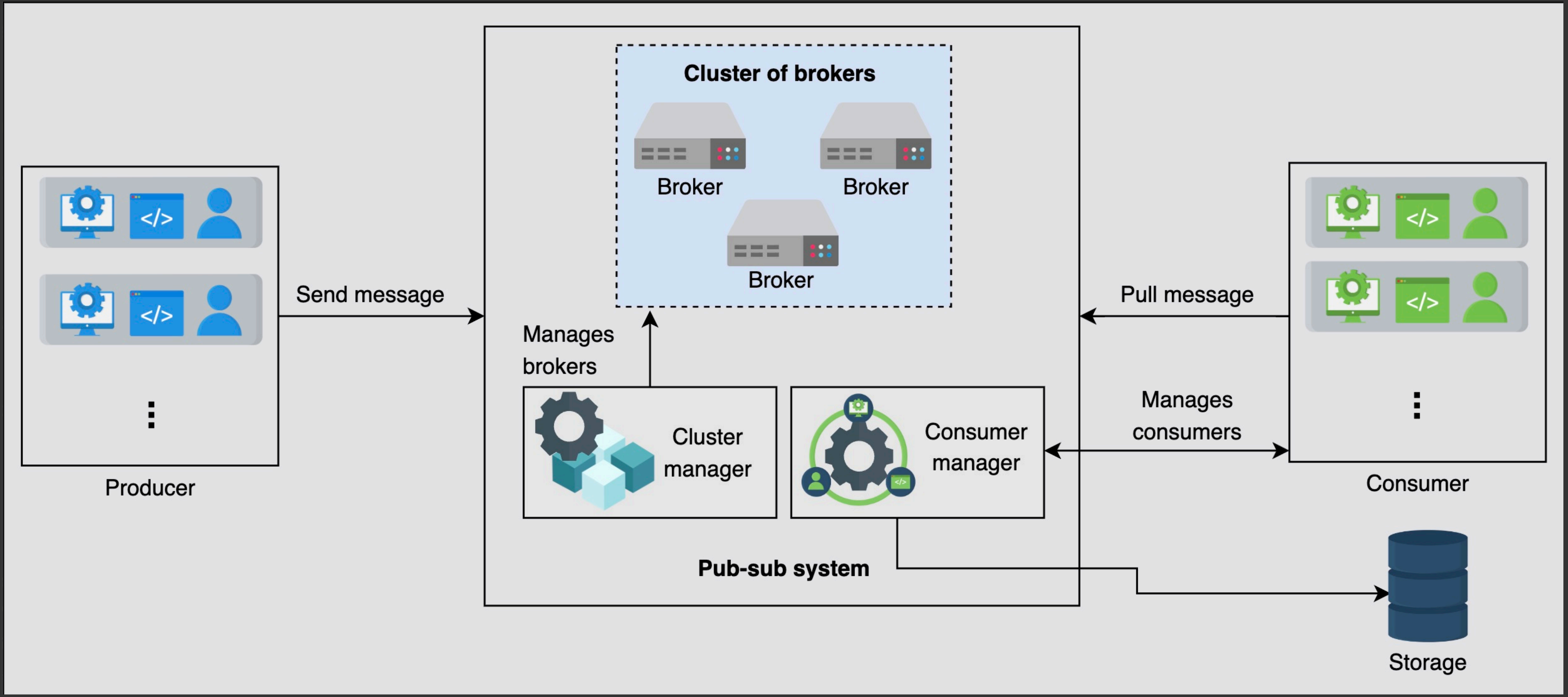
Cluster manager: We’ll have numerous broker servers to cater to our scalability needs. We need a cluster manager to supervise the broker’s health. It will notify us if a broker fails.

Storage: We’ll use a relational database to store consumer details, such as subscription information and retention period.

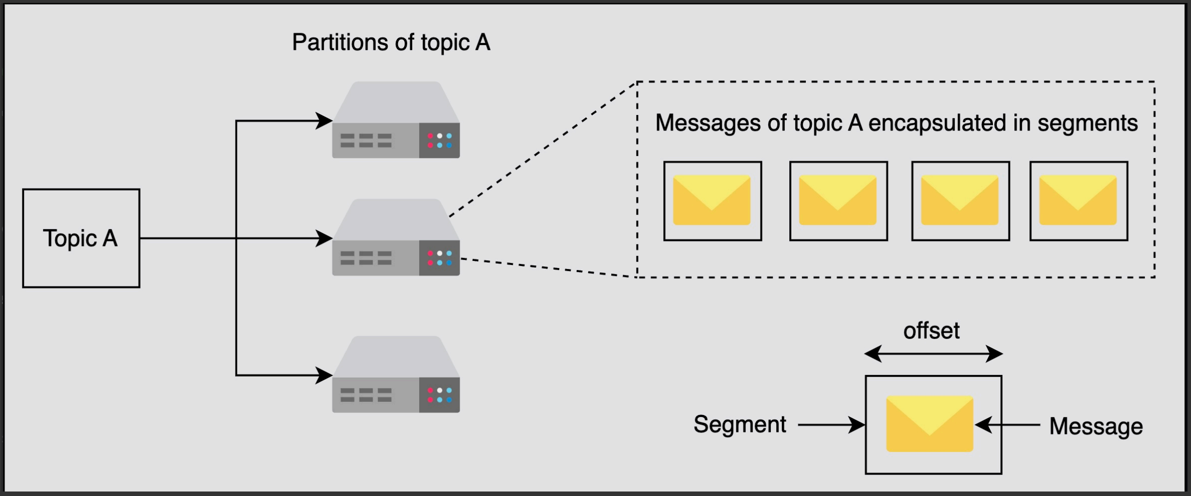
Consumer manager: This is responsible for managing the consumers. For example, it will verify if the consumer is authorized to read a message from a certain topic or not.

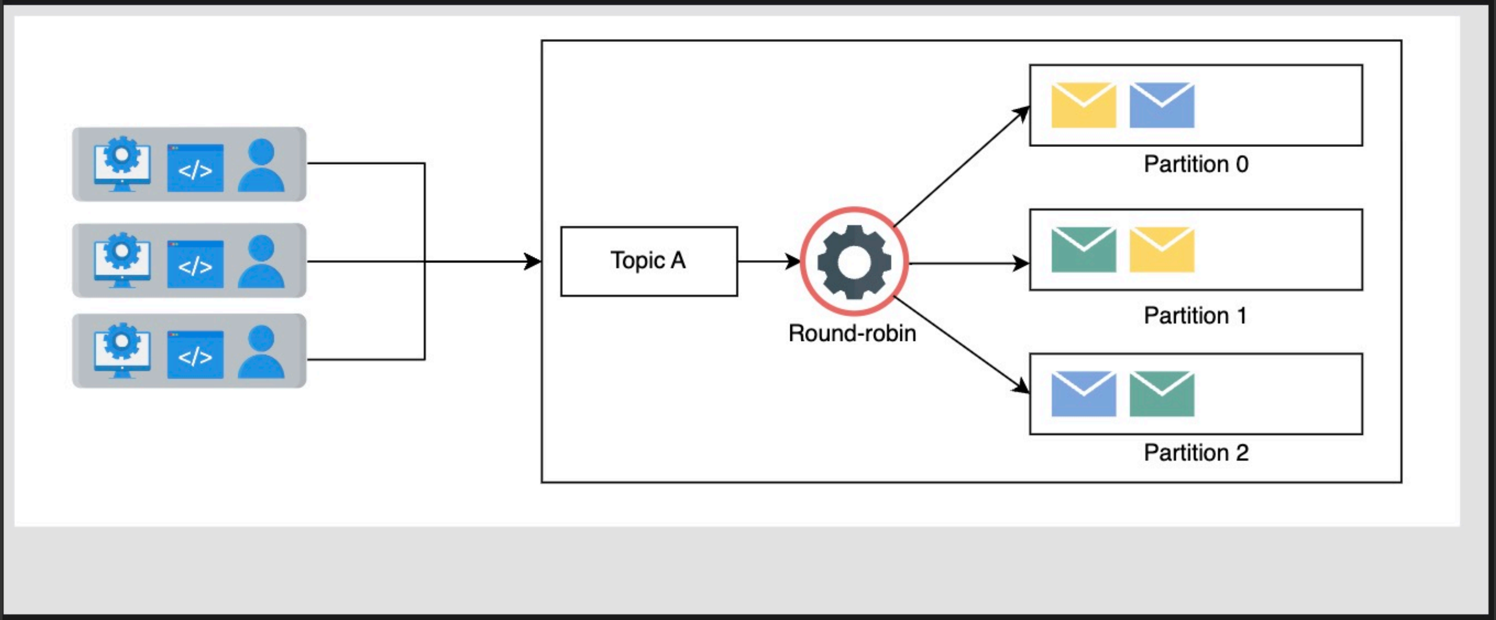
Acknowledgment: An acknowledgment is used to notify the producer that the received message has been stored successfully. The system will wait for an acknowledgment from the consumer if it has successfully consumed the message.

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.



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.


Partitions of topic A
Topic A
Messages of topic A encapsulated in segments
offset
Segment
Message


Topic A
Round-robin
Partition 0
Partition 1
Partition 2

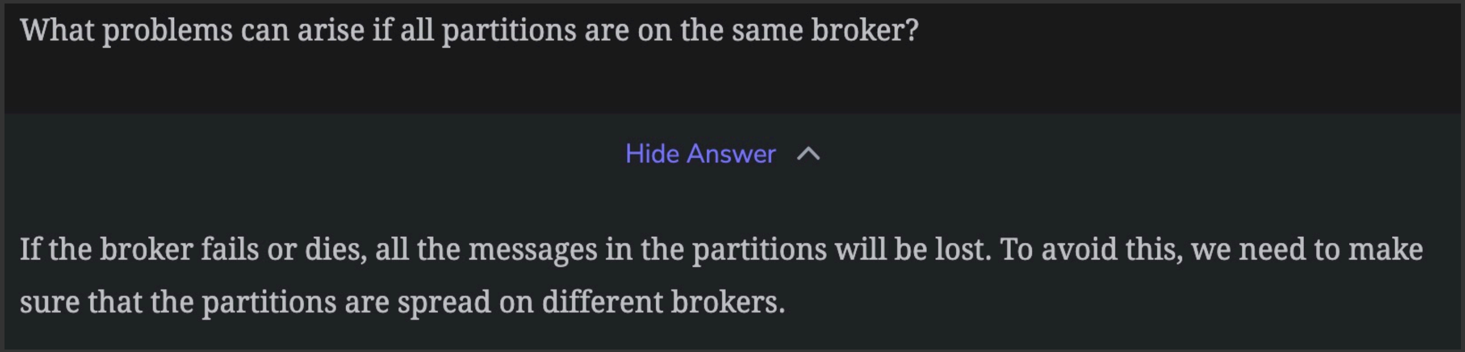
Strict ordering ensures that the messages are stored in the order in which they are produced. How can we ensure strict ordering for our messages?

Hide Answer
We'll assign each partition a unique ID, partition_ID. The user can provide the partition_ID while writing into the system. In this way, the messages will be sent to the specified partition, and the ordering will be strict. Our API call to write into the pub-sub system looks like this:

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

If the user does not provide the partition_ID, we'll use the weighted round-robin algorithm to decide which message has to be sent to which partition.

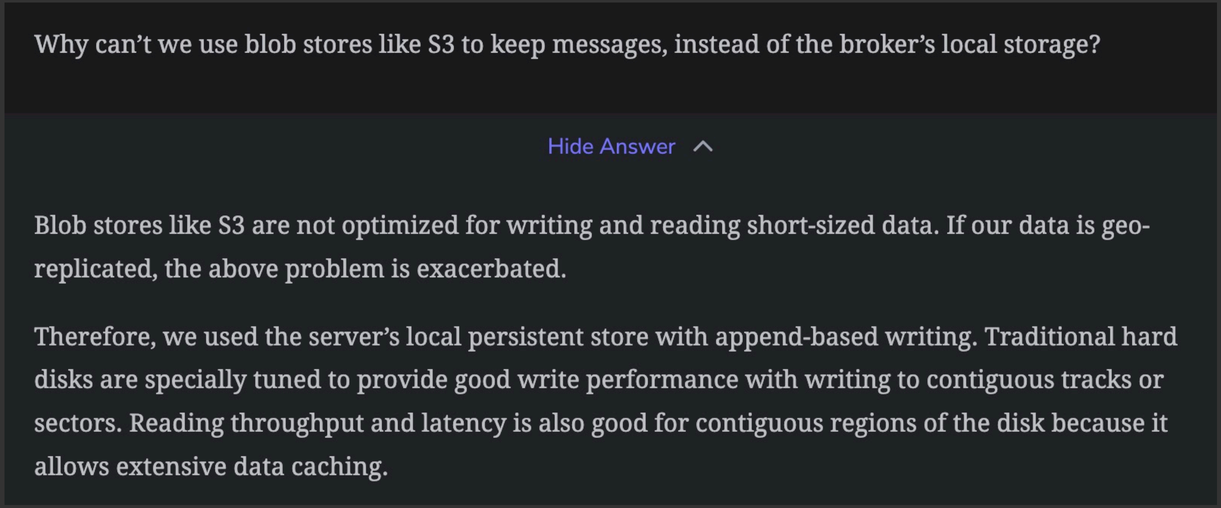
It might seem strange to give the ability to choose a partition to the client of pub-sub. However, such a facility can be the basis from where clients can get data for some specific time period—for example, getting data from yesterday. For simplicity, we won't include time-based reading in our design.



What problems can arise if all partitions are on the same broker?

Hide Answer

If the broker fails or dies, all the messages in the partitions will be lost. To avoid this, we need to make sure that the partitions are spread on different brokers.

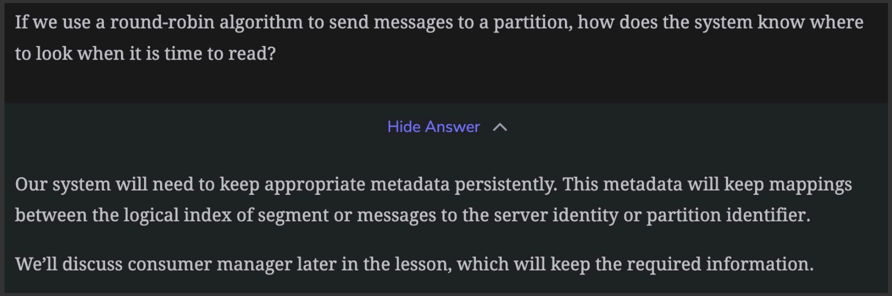


Why can't we use blob stores like S3 to keep messages, instead of the broker's local storage?

Hide Answer

Blob stores like S3 are not optimized for writing and reading short-sized data. If our data is geo-replicated, the above problem is exacerbated.

Therefore, we used the server's local persistent store with append-based writing. Traditional hard disks are specially tuned to provide good write performance with writing to contiguous tracks or sectors. Reading throughput and latency is also good for contiguous regions of the disk because it allows extensive data caching.

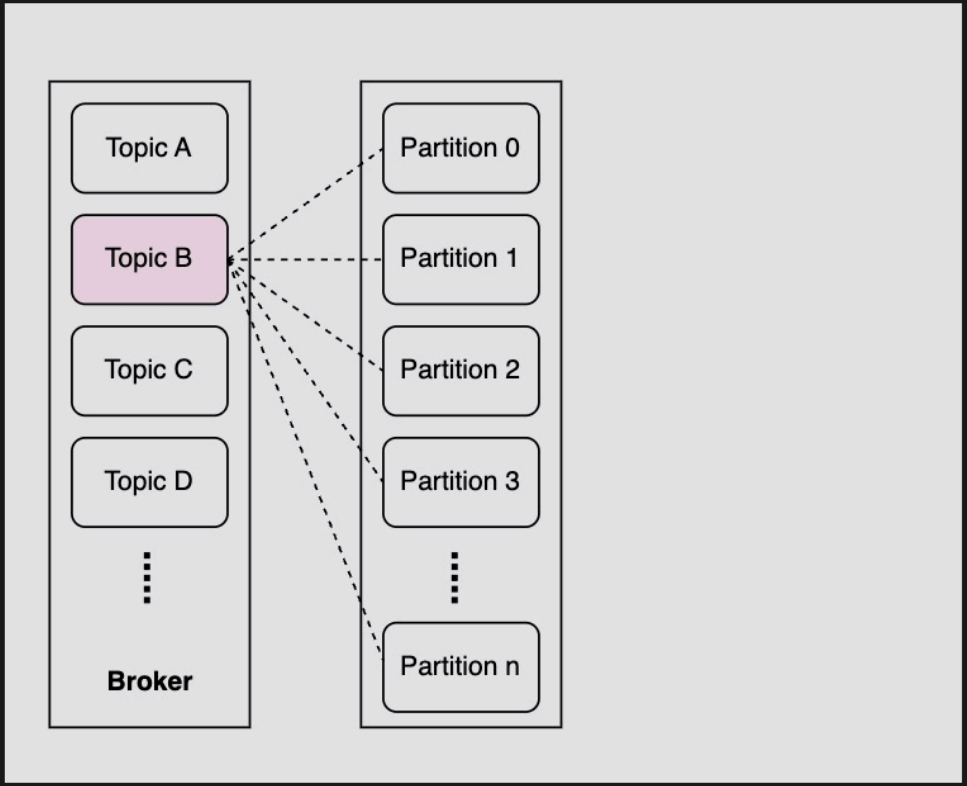


If we use a round-robin algorithm to send messages to a partition, how does the system know where to look when it is time to read?

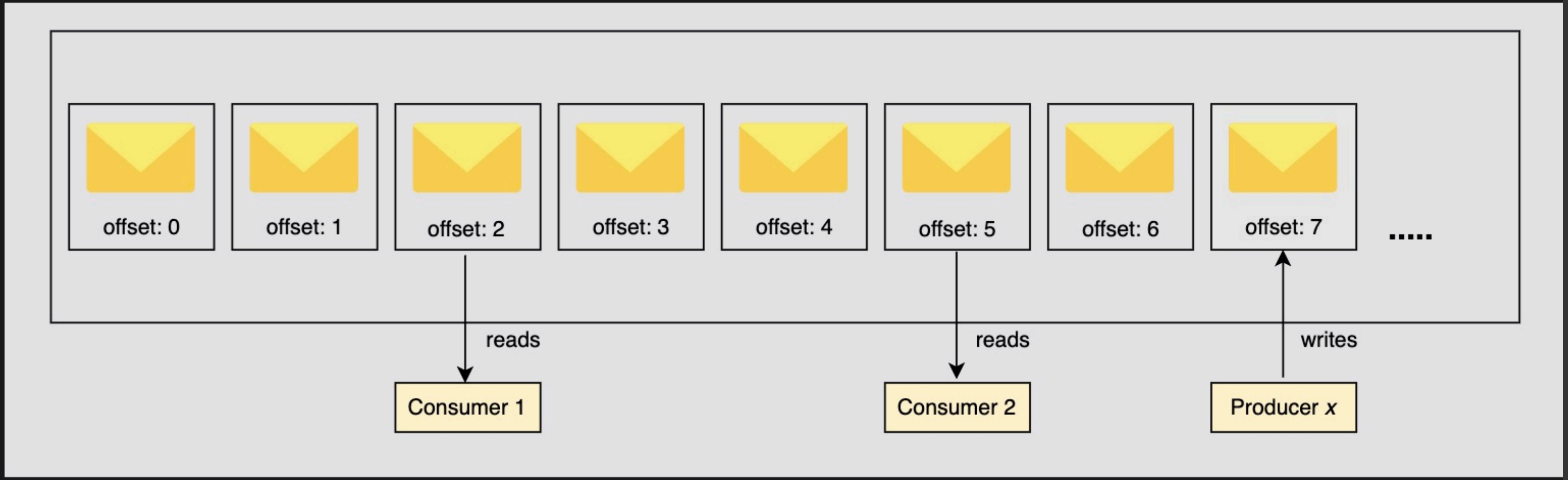
Hide Answer

Our system will need to keep appropriate metadata persistently. This metadata will keep mappings between the logical index of segment or messages to the server identity or partition identifier.

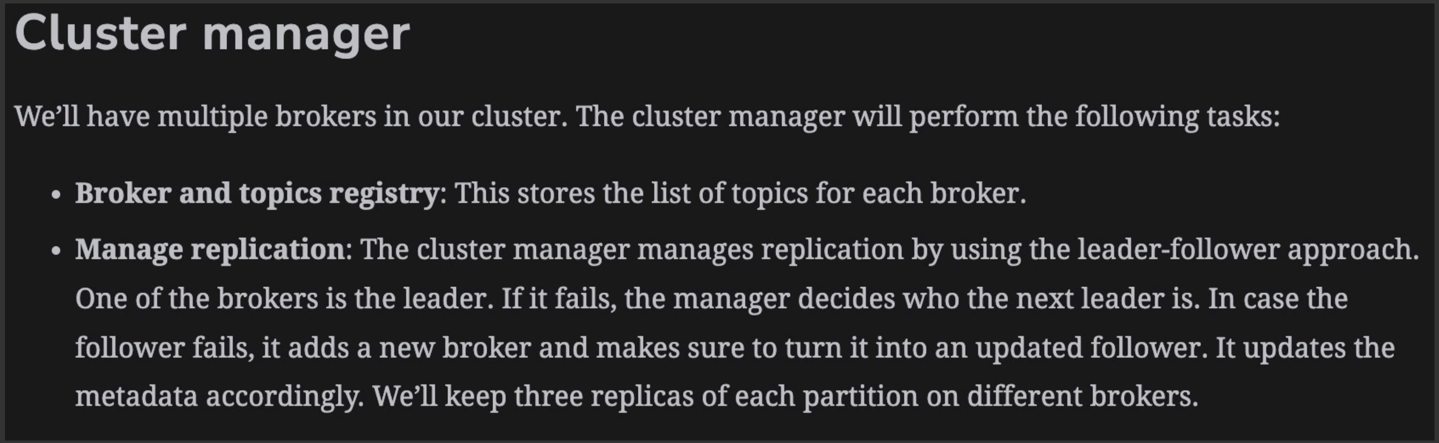
We'll discuss consumer manager later in the lesson, which will keep the required information.



an represent the partitions in another way. Here, Topic B is split into multiple partitions



The consumers can read anywhere from the file. Producers add to the end of file



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.

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.

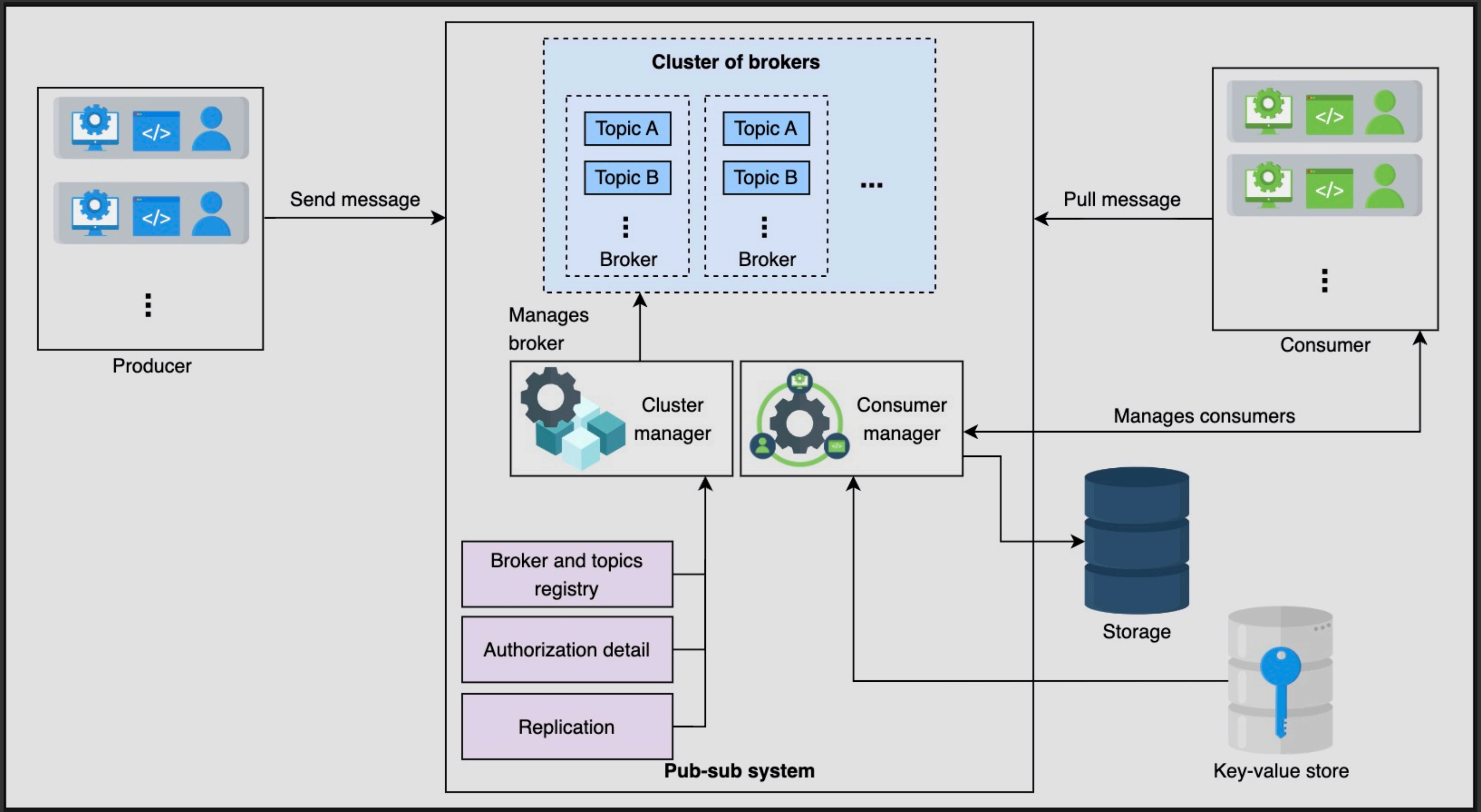
Retention time management: The manager will also verify if the consumer is allowed to read the specific message or not. If, according to its retention time, the message should be inaccessible to the consumer, then it will not allow the consumer to read the message.

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



Finalized design