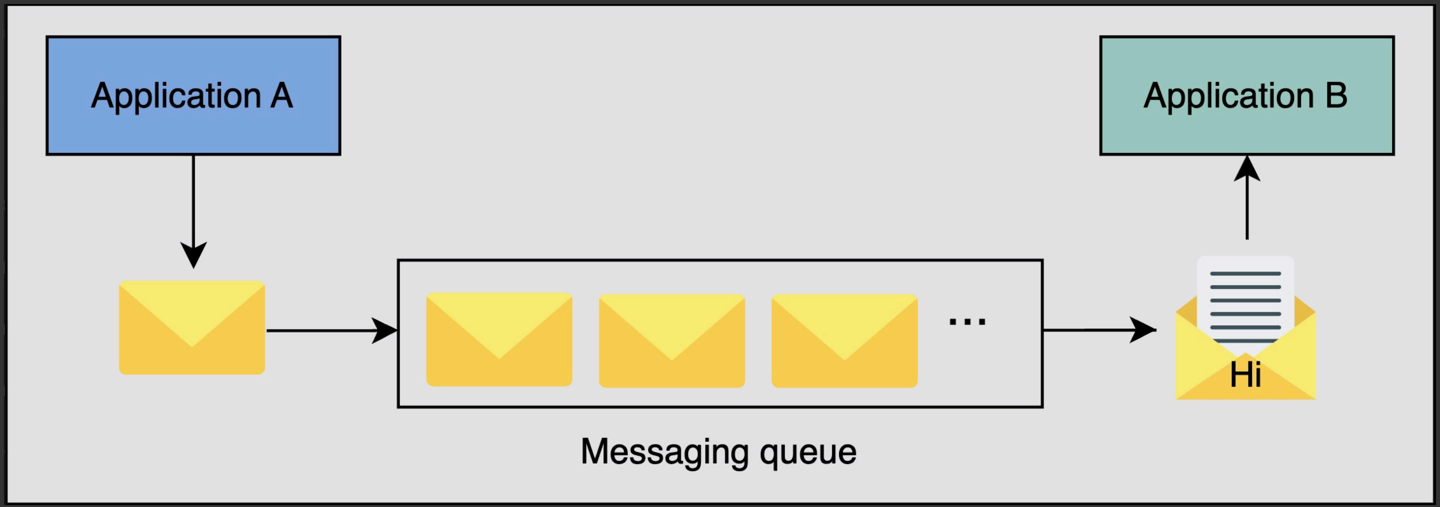
What is a messaging Queue?

→ A messaging queue in an intermediate component between the interacting entities known as producers & consumers.



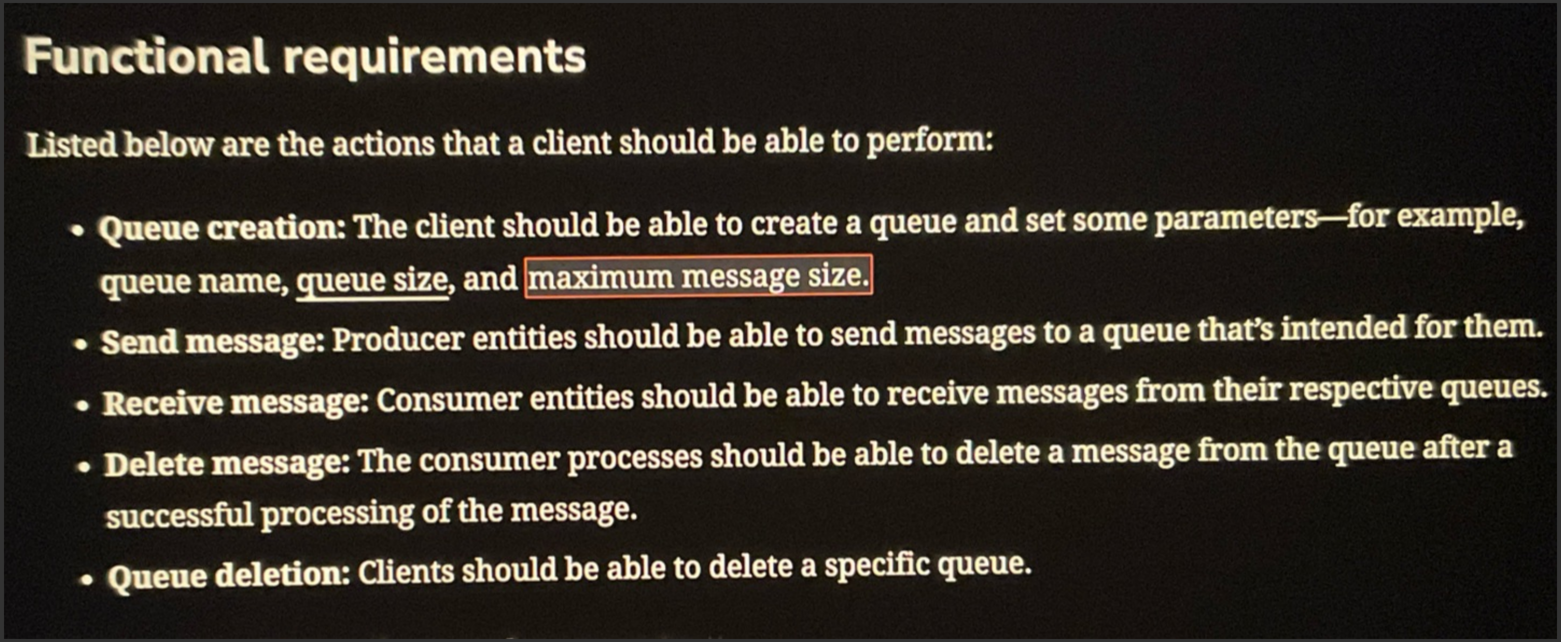
Benefits of messaging Queue

- → Improved performance
- → Better Reliability
- → Granular scalability
- → Easy decoupling
- → Rate limiting
- → Priority Queue.

Messaging queue use cases

1. Sending many email
2. Data post-processing
3. Recommender system.

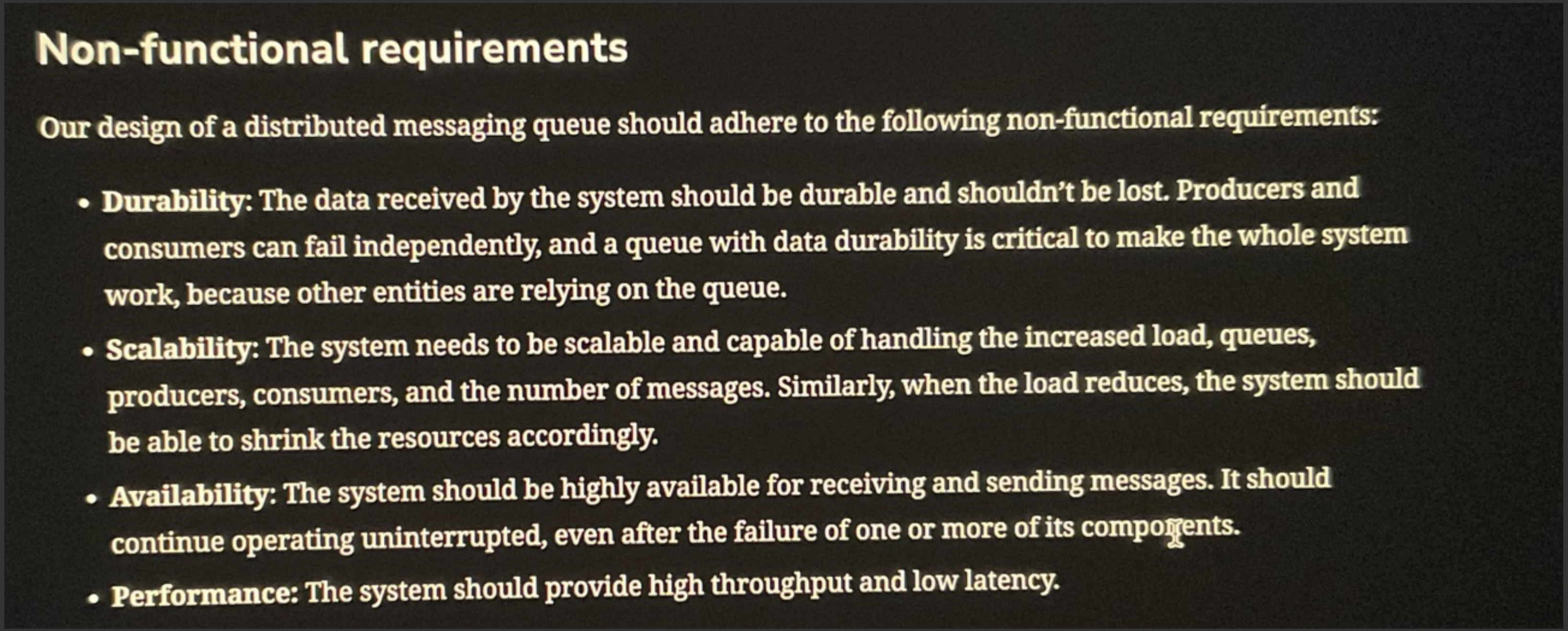
Requirement of a Distributed Messaging Queue Design



Functional requirements

Listed below are the actions that a client should be able to perform:

- **Queue creation:** The client should be able to create a queue and set some parameters—for example, queue name, queue size, and maximum message size.
- **Send message:** Producer entities should be able to send messages to a queue that's intended for them.
- **Receive message:** Consumer entities should be able to receive messages from their respective queues.
- **Delete message:** The consumer processes should be able to delete a message from the queue after a successful processing of the message.
- **Queue deletion:** Clients should be able to delete a specific queue.

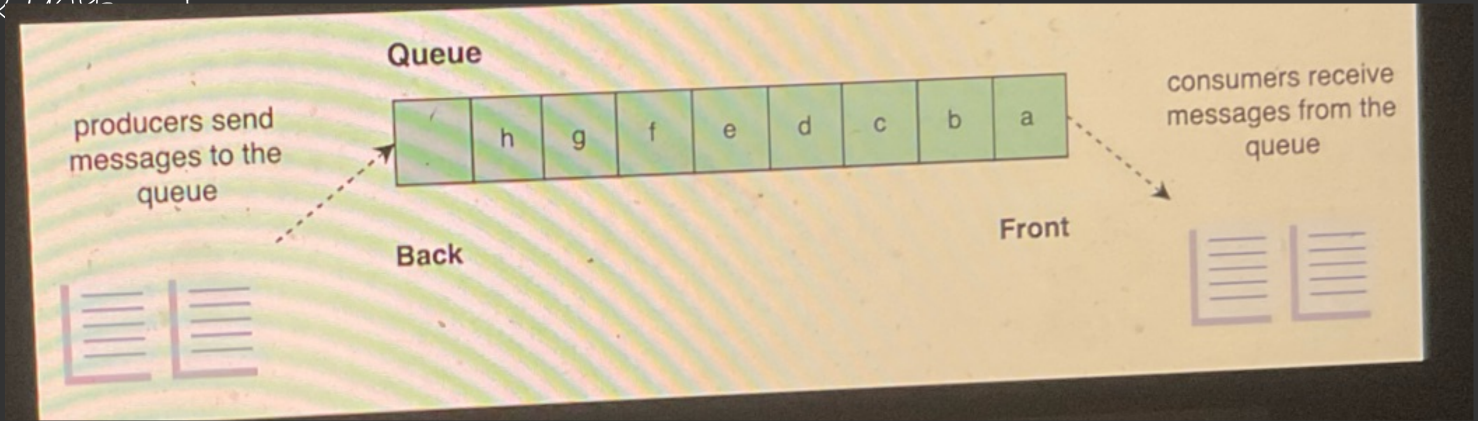


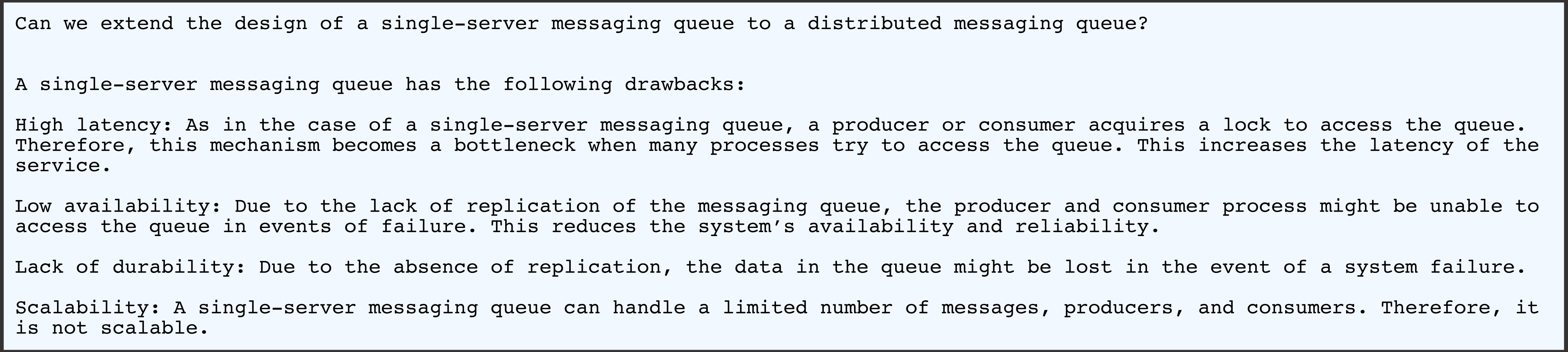
Non-functional requirements

Our design of a distributed messaging queue should adhere to the following non-functional requirements:

- **Durability:** The data received by the system should be durable and shouldn't be lost. Producers and consumers can fail independently, and a queue with data durability is critical to make the whole system work, because other entities are relying on the queue.
- **Scalability:** The system needs to be scalable and capable of handling the increased load, queues, producers, consumers, and the number of messages. Similarly, when the load reduces, the system should be able to shrink the resources accordingly.
- **Availability:** The system should be highly available for receiving and sending messages. It should continue operating uninterrupted, even after the failure of one or more of its components.
- **Performance:** The system should provide high throughput and low latency.

Queue are used within a single server where the producer & consumer processes are also on the same node. A producer or consumer can access a single-server queue by acquiring the locking mechanism to avoid data inconsistency.


Queue
producers send messages to the queue
h
g
f
e
d
c
b
a
consumers receive messages from the queue
Back
Front



Can we extend the design of a single-server messaging queue to a distributed messaging queue?

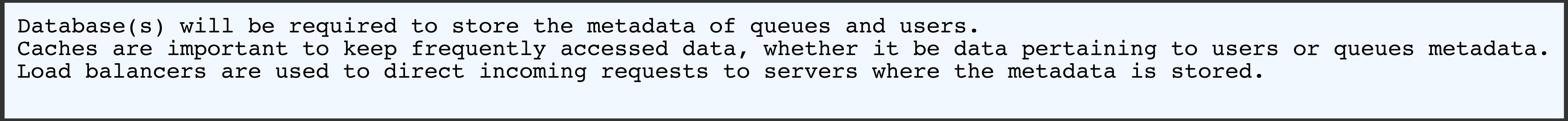
A single-server messaging queue has the following drawbacks:

High latency: As in the case of a single-server messaging queue, a producer or consumer acquires a lock to access the queue. Therefore, this mechanism becomes a bottleneck when many processes try to access the queue. This increases the latency of the service.

Low availability: Due to the lack of replication of the messaging queue, the producer and consumer process might be unable to access the queue in events of failure. This reduces the system’s availability and reliability.

Lack of durability: Due to the absence of replication, the data in the queue might be lost in the event of a system failure.

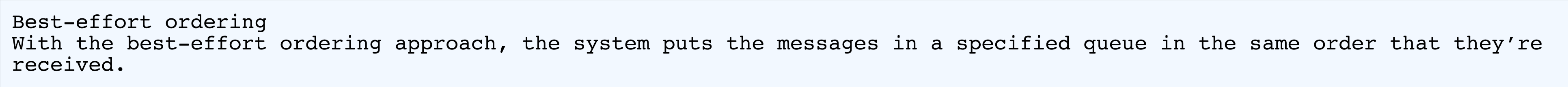
Scalability: A single-server messaging queue can handle a limited number of messages, producers, and consumers. Therefore, it is not scalable.



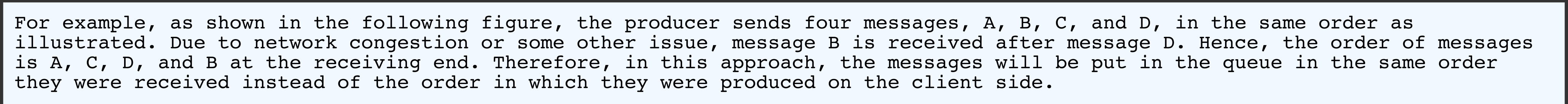
Database(s) will be required to store the metadata of queues and users.
Caches are important to keep frequently accessed data, whether it be data pertaining to users or queues metadata.
Load balancers are used to direct incoming requests to servers where the metadata is stored.

Ordering of messages

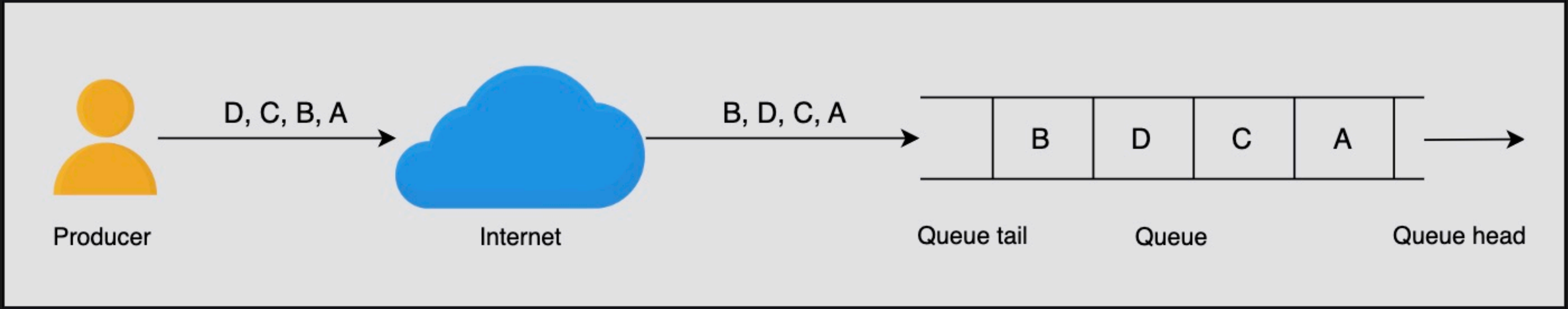
A messaging queue is used to receive messages from producers. These messages are consumed by the consumers at their own pace.



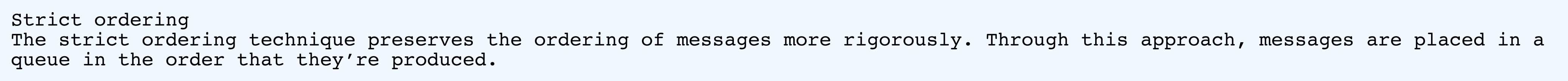
Best-effort ordering
With the best-effort ordering approach, the system puts the messages in a specified queue in the same order that they're received.



For example, as shown in the following figure, the producer sends four messages, A, B, C, and D, in the same order as illustrated. Due to network congestion or some other issue, message B is received after message D. Hence, the order of messages is A, C, D, and B at the receiving end. Therefore, in this approach, the messages will be put in the queue in the same order they were received instead of the order in which they were produced on the client side.



Best-effort ordering: Messages are placed in a queue in the same order that they're received and not in the order they were sent



Strict ordering
The strict ordering technique preserves the ordering of messages more rigorously. Through this approach, messages are placed in a queue in the order that they're produced.

Three approaches can be used for ordering incoming messages.

→ Monotonically increasing numbers

→ Causality - based sorting at the server side

→ Using time stamp for synchronized clocks.

Suppose that a message sent earlier arrives late due to a network delay. What would be the proper approach to handle such a situation?

Hide Answer

The simple solution in such cases is to reorder the queue. Two scenarios can arise from this. First, reordering puts the messages in the correct order. Second, we’ve already handed out newer messages to the consumers.

If an old message comes when we’ve already handed out a newer message, we put it in a special queue, and the client handles that situation. The client may later decide whether to consume the message if the message does not affect the intended operation or discard it if it’s not needed.

Effect on performance

Primarily, a queue is designed for first-in, first-out (FIFO) operations;. First-in, first-out operations suggest that the first message that enters a queue is always handed out first. However, it isn’t easy to maintain this strict order in distributed systems. Since message A was produced before message B, it’s still uncertain that message A will be consumed before message B. Using monotonically increasing message identifiers or causality-bearing identifiers provide high throughput while putting messages in a queue. Though the need for the online sorting to provide a strict order takes some time before messages are ready for extraction. To minimize latency caused by the online sorting, we use a time-window approach.

Managing concurrency

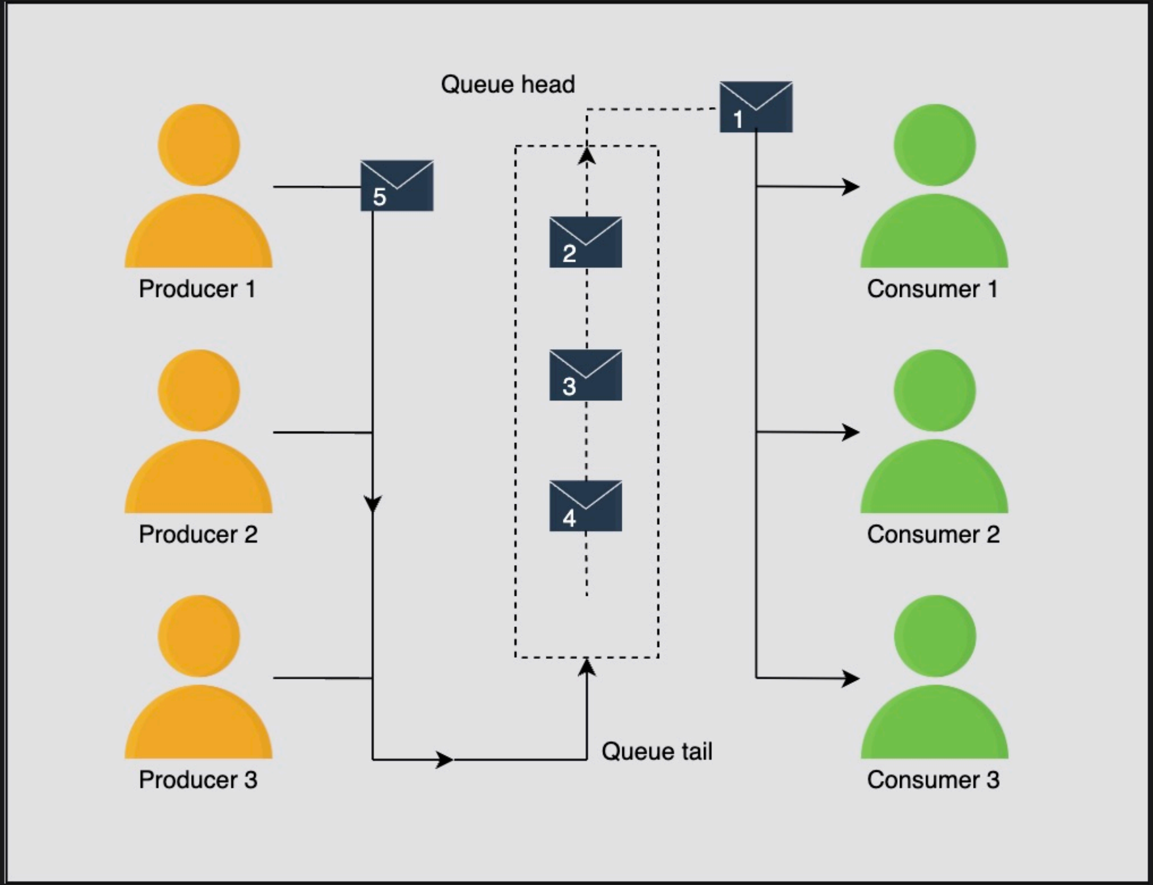
Concurrent queue access needs proper management. Concurrency can take place at the following stages:

When multiple messages arrive at the same time.

When multiple consumers request concurrently for a message.

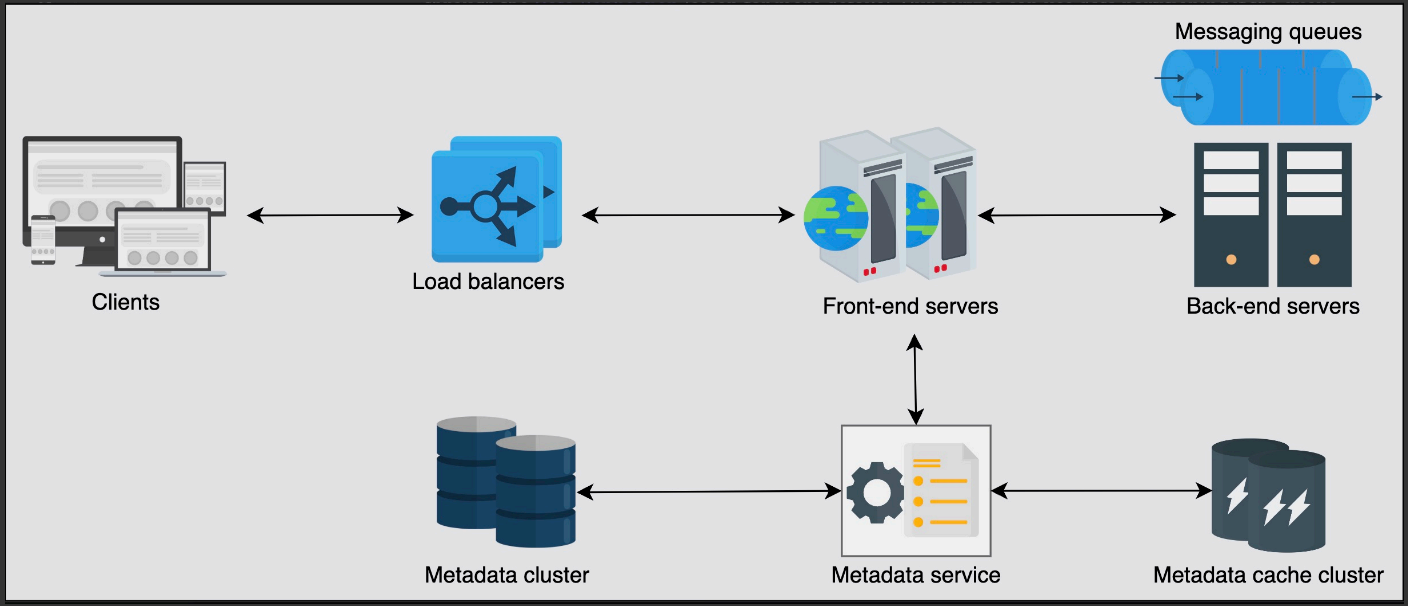
The first solution is to use the locking mechanism. When a process or thread requests a message, it should acquire a lock for placing or consuming messages from the queue. However, as was discussed earlier, this approach has several drawbacks. It’s neither scalable nor performant.

Another solution is to serialize the requests using the system’s buffer at both ends of the queue so that the incoming messages are placed in an order, and consumer processes also receive messages in their arrival sequence. By serializing requests, we mean that the requests (either for putting data or extracting data), which come to the server would be queued by the OS, and a single application thread will put them in the queue (we can assume that both kinds of requests, put and extract come to the same port) without any locking. It will be a possible lock-free solution, providing high throughput. This is a more viable solution because it can help us avoid the occurrence of race conditions.



Avoiding race conditions: Producers and consumers are serialized at both ends of the queue

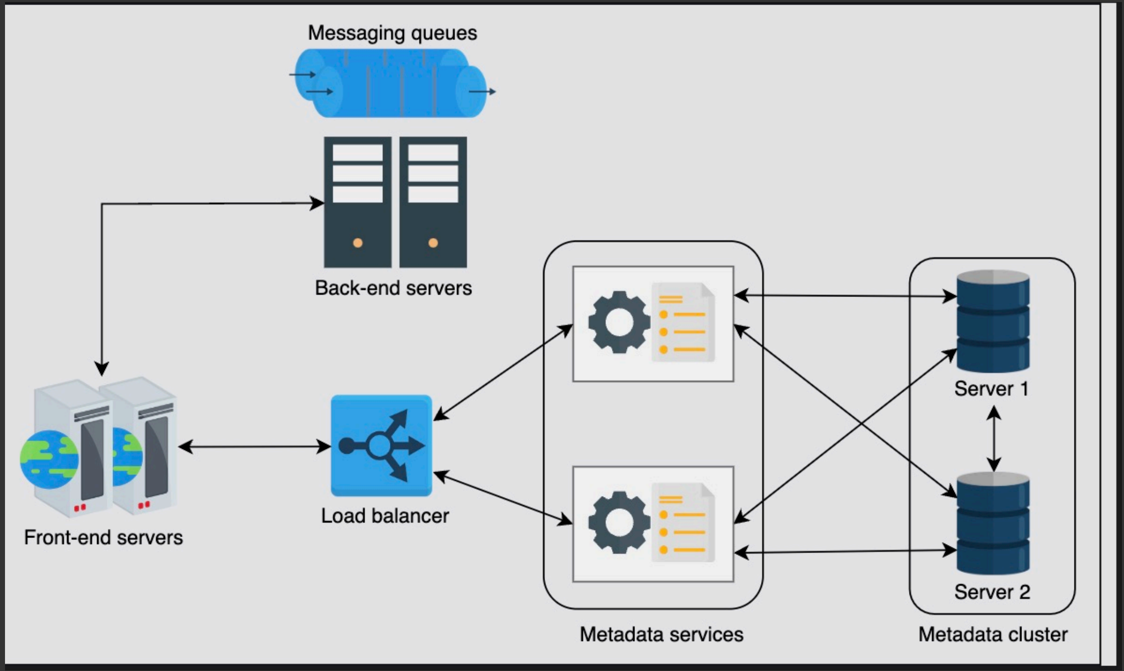
Design a distributed Messaging Queue

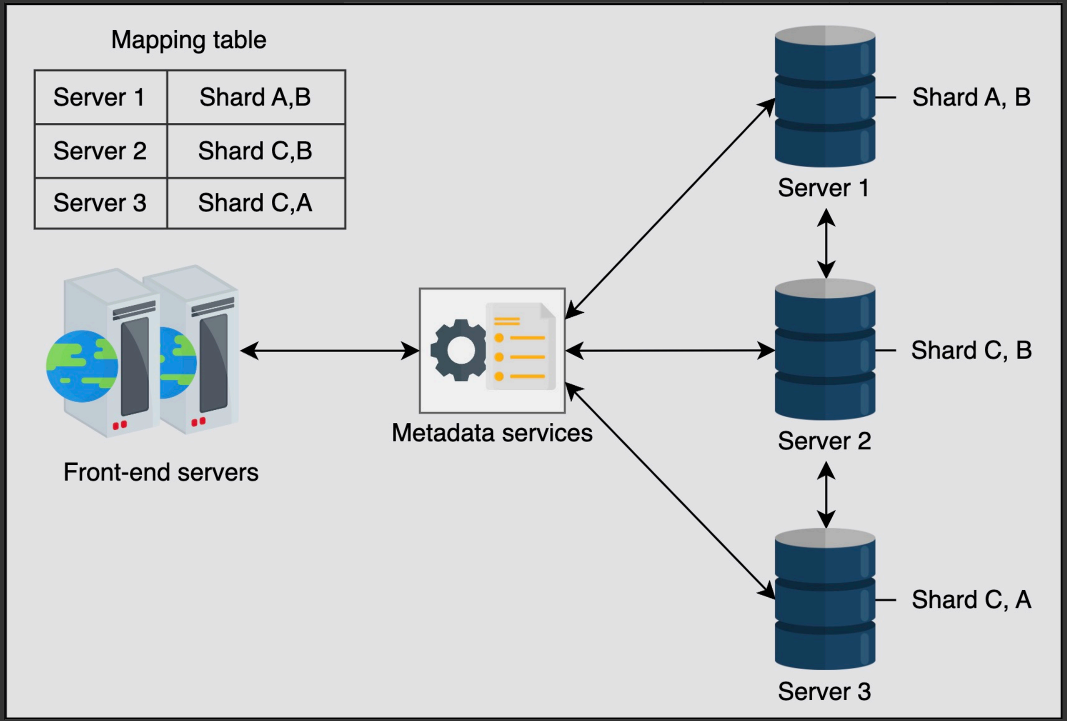


Load balancer
The load balancer layer receives requests from producers and consumers, which are forwarded to one of the front-end servers.

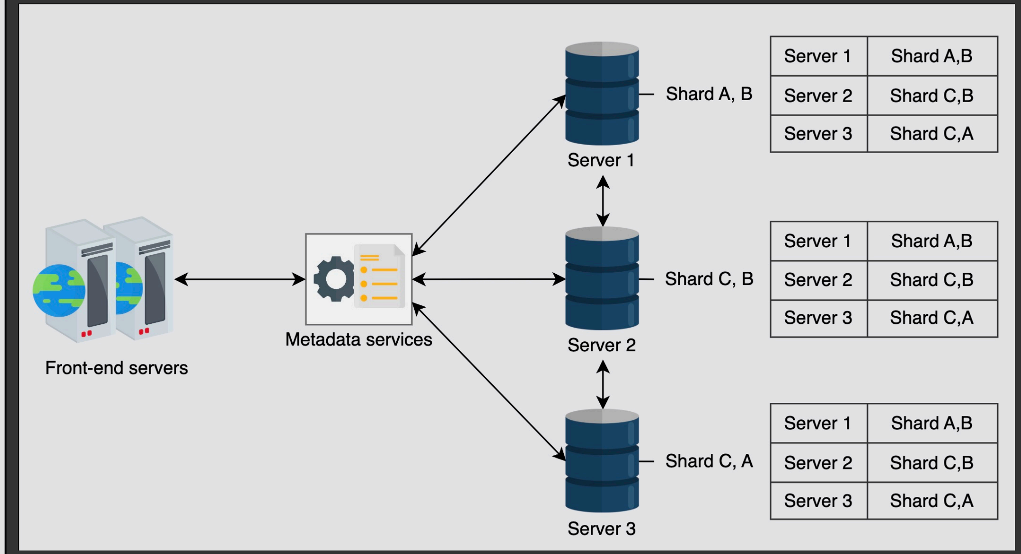
Front-end service:
Request validation
Authentication and authorization
Caching
Request Dispatching
Request Deduplication
Usage data Collection

Metadata service
This component is responsible for storing, retrieving, and updating the metadata of queues in the metadata store and cache.




Mapping table

Server 1	Shard A,B
Server 2	Shard C,B
Server 3	Shard C,A


Server 1	Shard A,B
Server 2	Shard C,B
Server 3	Shard C,A

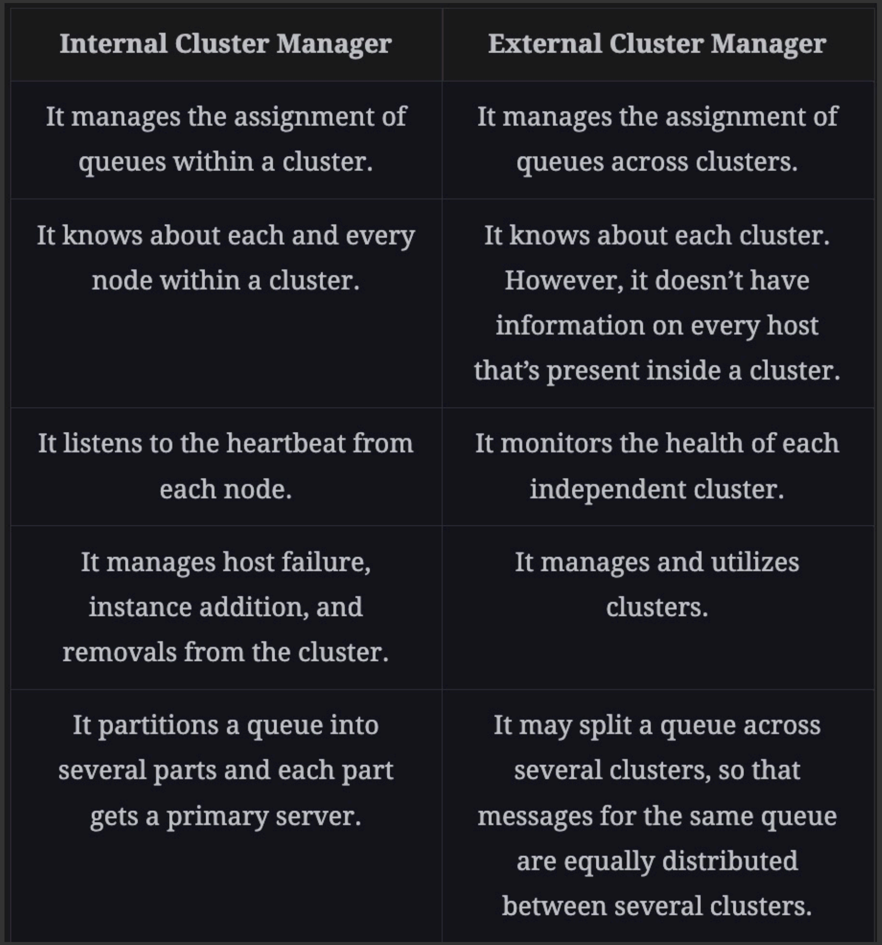
Server 1	Shard A,B
Server 2	Shard C,B
Server 3	Shard C,A

Server 1	Shard A,B
Server 2	Shard C,B
Server 3	Shard C,A

Back-end service

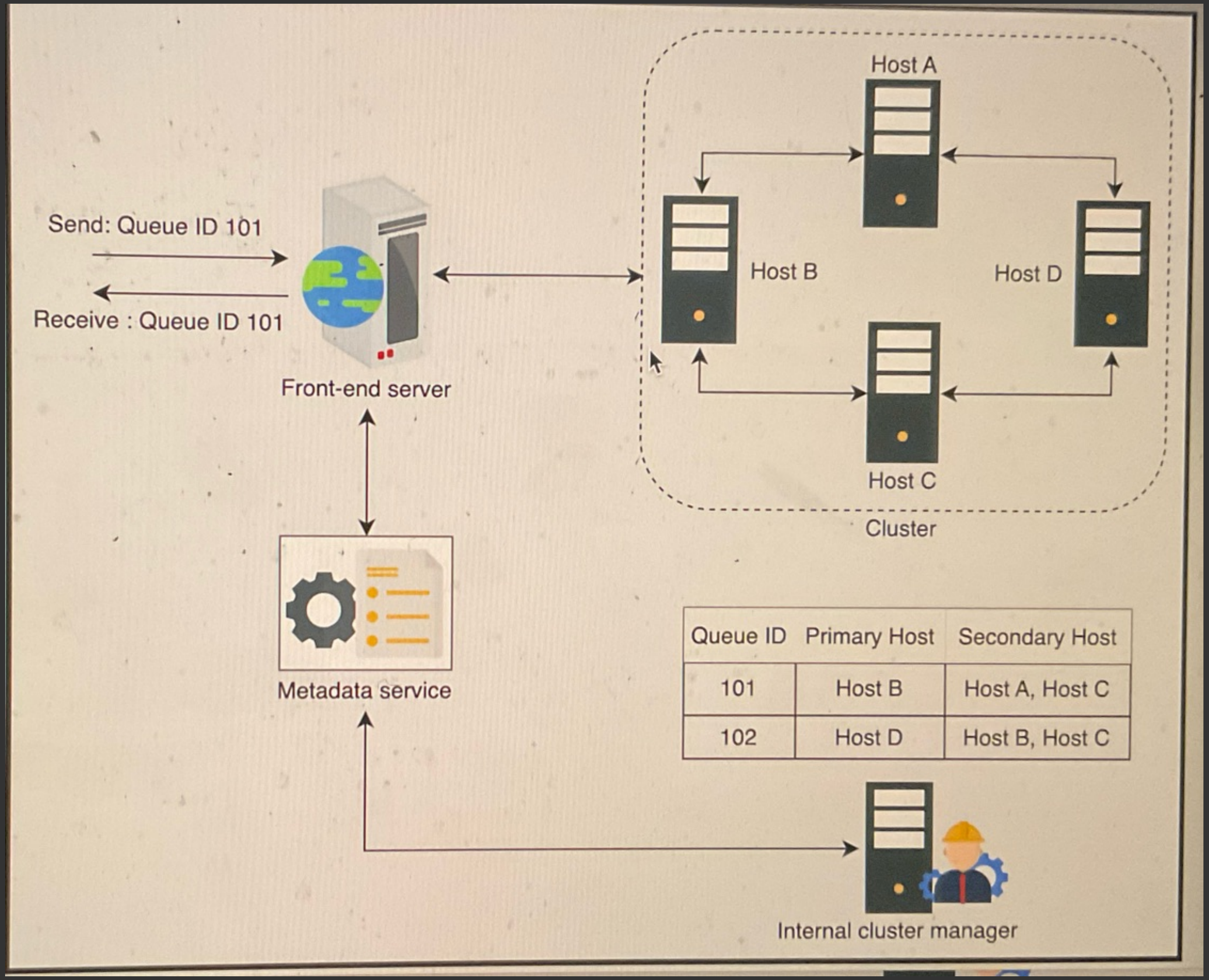
This is the core part of the architecture where major activities take place. When the front-end receives a message, it refers to the metadata service to determine the host where the message needs to be sent. The message is then forwarded to the host and is replicated on the relevant hosts to overcome a possible availability issue

Primary-secondary model
A cluster of independent hosts

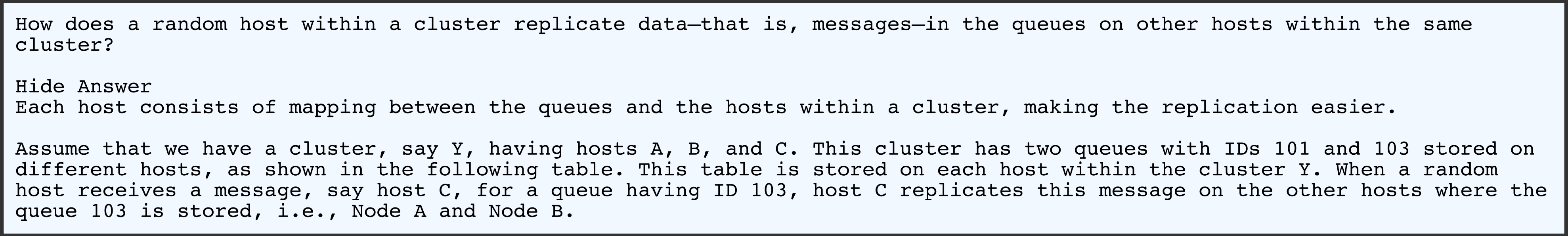


Internal Cluster Manager	External Cluster Manager
It manages the assignment of queues within a cluster.	It manages the assignment of queues across clusters.
It knows about each and every node within a cluster.	It knows about each cluster. However, it doesn’t have information on every host that’s present inside a cluster.
It listens to the heartbeat from each node.	It monitors the health of each independent cluster.
It manages host failure, instance addition, and removals from the cluster.	It manages and utilizes clusters.
It partitions a queue into several parts and each part gets a primary server.	It may split a queue across several clusters, so that messages for the same queue are equally distributed between several clusters.

```
For example, suppose we have two queues with the identities 101 and 102 residing on four different hosts A, B, C, and D. In this
example, instance B is the primary host of queue 101 and the secondary hosts where the queue 101 is replicated are A and C.
```



Queue ID	Primary Host	Secondary Host
101	Host B	Host A, Host C
102	Host D	Host B, Host C

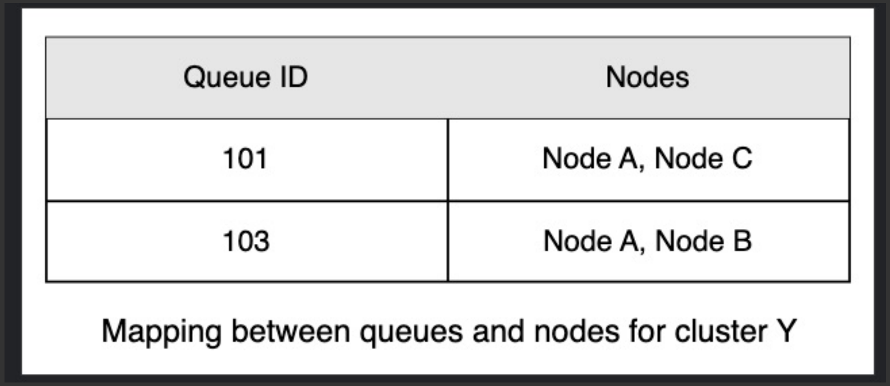


How does a random host within a cluster replicate data—that is, messages—in the queues on other hosts within the same cluster?

Hide Answer

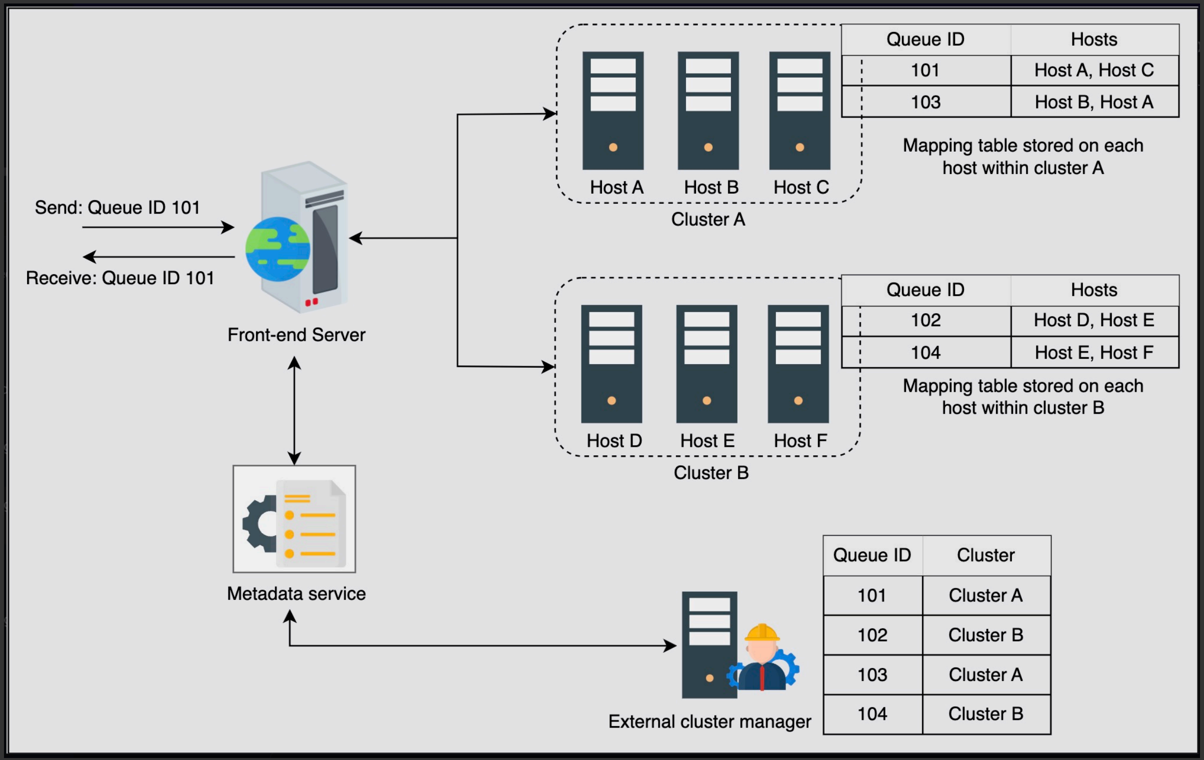
Each host consists of mapping between the queues and the hosts within a cluster, making the replication easier.

Assume that we have a cluster, say Y, having hosts A, B, and C. This cluster has two queues with IDs 101 and 103 stored on different hosts, as shown in the following table. This table is stored on each host within the cluster Y. When a random host receives a message, say host C, for a queue having ID 103, host C replicates this message on the other hosts where the queue 103 is stored, i.e., Node A and Node B.



Queue ID	Nodes
101	Node A, Node C
103	Node A, Node B

Mapping between queues and nodes for cluster Y



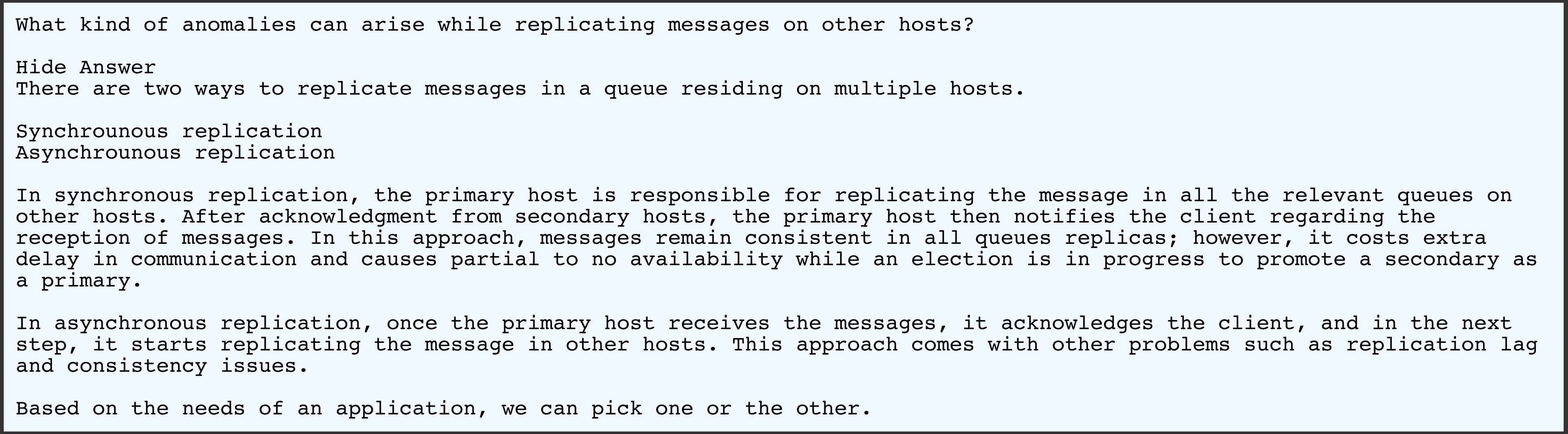
Queue ID	Hosts
101	Host A, Host C
103	Host B, Host A

Mapping table stored on each host within cluster A

Queue ID	Hosts
102	Host D, Host E
104	Host E, Host F

Mapping table stored on each host within cluster B

Queue ID	Cluster
101	Cluster A
102	Cluster B
103	Cluster A
104	Cluster B



What kind of anomalies can arise while replicating messages on other hosts?

Hide Answer
There are two ways to replicate messages in a queue residing on multiple hosts.

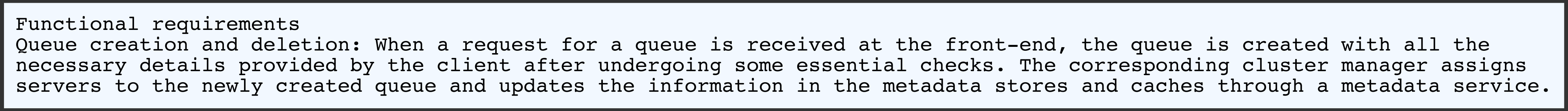
Synchrounous replication
Asynchrounous replication

In synchronous replication, the primary host is responsible for replicating the message in all the relevant queues on other hosts. After acknowledgment from secondary hosts, the primary host then notifies the client regarding the reception of messages. In this approach, messages remain consistent in all queues replicas; however, it costs extra delay in communication and causes partial to no availability while an election is in progress to promote a secondary as a primary.

In asynchronous replication, once the primary host receives the messages, it acknowledges the client, and in the next step, it starts replicating the message in other hosts. This approach comes with other problems such as replication lag and consistency issues.

Based on the needs of an application, we can pick one or the other.

Evaluation of Distributed Messaging queue



Functional requirements
Queue creation and deletion: When a request for a queue is received at the front-end, the queue is created with all the necessary details provided by the client after undergoing some essential checks. The corresponding cluster manager assigns servers to the newly created queue and updates the information in the metadata stores and caches through a metadata service.

How do we handle messages that can’t be processed—here meaning consumed—after maximum processing attempts by the consumer?

A special type of queue, called a dead-letter queue, can be provided to handle messages that aren’t consumed after the maximum number of processing attempts have been made by the consumer. This type of queue is also used for keeping messages that can’t be processed successfully due to the following factors:

The messages intended for a queue that doesn’t exist anymore.
The queue length limit is exceeded, although this would rarely occur with our current design.
The message expires due to per-message time to live (TTL).
A dead-letter queue is also important for determining the cause of failure and for identifying faults in the system.

What happens when the visibility timeout of a specific message expires and the consumer is still busy processing the message?

Hide Answer
The message becomes visible, and another worker can receive the message, thereby duplicating the processing. To avoid such a situation, we ensure that the application sets a safe threshold for visibility timeout.

Non-functional requirements
Durability: To achieve durability, the queues’ metadata is replicated on different nodes. Similarly, when a message is received, it’s replicated in the queues that reside on different nodes. Therefore, if a node fails, other nodes can be used to deliver or retrieve messages.

Scalability: Our design components, such as front-end servers, metadata servers, caches, back-end clusters, and more are horizontally scalable. We can add to or remove their capacity to match our needs. The scalability can be divided into two dimensions:

Increase in the number of messages: When the number of messages touches a specific limit—say, 80%—the specified queue is expanded. Similarly, the queue is shrunk when the number of messages drops below a certain threshold.

Increase in the number of queues: With an increasing number of queues, the demand for more servers also increases, in which case the cluster manager is responsible for adding extra servers. We commission nodes so that there is performance isolation between different queues. An increased load on one queue shouldn’t impact other queues.

Availability: Our data components, metadata and actual messages, are properly replicated inside or outside the data center, and the load balancer routes traffic around failed nodes. Together, these mechanisms make sure that our system remains available for service under faults.

Performance: For better performance we use caches, data replication, and partitioning, which reduces the data reads and writes time. Moreover, the best effort ordering strategy for ordering messages is there to use to increase the throughput and lower the latency when it’s necessary. In the case of strict ordering, we also suggest time-window based sorting to potentially reduce the latency.