Evaluation of a Distributed Messaging Queue's Design

Evaluate the proposed system based on the functional and non-functional requirements of a distributed messaging queue.

We'll cover the following

We completed the process of designing a distributed messaging queue. Now, let's analyze whether the design met the functional and non-functional requirements of a 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.

  Similarly, the queue is deleted when the client doesn't need it anymore. The responsible cluster manager deallocates the space occupied by the queue and, consequently, deletes the data from all the metadata stores and caches.

Point to Ponder

Question

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

Show Answer

- **Send and receive messages:** Producers can deliver messages to specific queues once they are created. At the back-end, receiving messages are sorted based on time stamps to preserve their order and are placed in the queue. Similarly, a consumer can retrieve messages from a specified queue.

When a message is received from a producer for a specific queue, the front-end identifies the primary host or cluster, depending on the replication model, where the queue resides. The request is then forwarded to the corresponding entity and put in the queue.

- **Message deletion:** Primarily, two options are used to delete a message from a queue.

1. The first option is to not delete a message after it’s consumed. However, in this case, the consumer is responsible for keeping track of what’s consumed. For this, we need to maintain the order of messages in the queue and keep track of a message within a queue. A job can then delete the message when the expiration conditions are met. Apache Kafka mostly uses this idea where multiple processes can consume a message.

2. The second approach also doesn't delete a message after it's consumed. However, it's made invisible for some time via an attribute—for example, `visibility_timeout`. This way, the other consumers are unable to get messages that have already been consumed. The message is then deleted by the consumer via an API call.

In both cases, the message being retrieved by the consumer is only deleted by the consumer. The reason behind this is to provide high durability if a consumer can't process a message due to some failure. In such a case, in the absence of a delete call, the consumer can retrieve the message again when it comes back.

Moreover, this approach also provides at-least-once delivery semantic. For example, when a worker fails to process the message, another worker can retrieve the message after it becomes visible in the queue.

Point to Ponder

Question

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

Show Answer

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:

    Similarly, the queue is shrunk when the number of messages drops below a certain threshold.

2. **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.

Describe how a distributed messaging queue would be used in an online multiplayer game to manage real-time player actions (movement, attacks, etc.). What are the crucial requirements for this scenario?

Want to know the correct answer?

How can you use a distributed messaging queue in online games?

H1 H2 H3 | B I | | | |

Provide your answer here!

Conclusion

We discussed many subtleties in designing a FIFO queue in a distributed setting. We saw that there is a trade-off between strict message production, message extraction orders, and achievable throughput and latency. Relaxed ordering gives us a higher throughput and lower latency. Asking for strict ordering forces the system to do extra work to enforce wall-clock or causality-based ordering. We use different data stores with appropriate replication and partitioning to scale with data. This design exercise highlights that a construct, a producer-consumer queue, that's simple to realize in a single-OS based system becomes much more difficult in a distributed setting.