ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 19 Message Queues

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 28, 2024

Outline

Reading Assignment

- This lecture: Apache Kafka
  `https://kafka.apache.org/documentation/#design`
- Next Lecture: 9

Outline

Message Queues

Kafka

Message Queues

- A middleware to enable message communication between senders and receivers, e.g. a message broker.
  - Reduce coupling by removing the immediate connections between message senders and receivers.
  - Serve as a buffer to reduce impact of performance difference between senders and receivers when there is a burst of load.
- How massages are distributed among receivers for a queue?
  - Producer-Consumer: senders are producers generating jobs as messages, receivers are consumers taking jobs out of the queue to work on them – no two consumers work on the same job.
  - Publisher-Subscriber (Pub-Sub): senders publish messages and all receivers as subscribers receive all messages.

Persistence

- What if queues fails?
- Persistence is required for producer-consumer queues.
  - Otherwise jobs may be lost.
- Persistence for Pub-Sub queues
  - With persistence, a subscriber can subscribe at any time to receive all past and future messages.
  - Without persistence, a subscriber only receives future messages after it subscribes but not past ones.
  - However, it takes resources to support persistence so one need to make a choice depending on application requirements.

Topic Management

- Queues are usually identified by topics.
  - Usually a meaningful string providing hints on what messages inside are all about.
  - More frequently used for Pub-Sub queues.
- Each publisher or sender, sends (topic, message) to the message broker.
  - So the message broker knows to which queue the message should go.
- Each subscriber or receiver, when establishing connection with the message broker, specifies what topics it is interested into.
  - Then the message broker will only send (topic, message) with matching topics to this subscriber.

Ideal Queue Behavior

- FIFO (first-in-first-out) ordering.
  - Messages are delivered to the receivers in the order they are sent by the senders.
- Exactly-once delivery.
  - Messages sent by senders are delivered to receivers exactly once – no lost messages and no repetition.
  - For producer-consumer queues, a producer generates a message and exactly one consumer consumes it exactly exactly once.
  - For publisher-subscriber queues, a publisher publishes a message and every subcriber receives it exactly once.
- Can we achieve these with networked services?
  - With a single message broker.
  - With multiple message brokers for horizontal scalability.

Communication Challenges for Ordering

- Consider when there is a single message broker.
  - The broker should be able to store and then send out messages in the order of their arrivals.
  - The FIFO ordering makes it possible to optimize persistence for high sequential read and write throughput.
- It takes time for messages to arrive from senders to the message broker and from the message broker to receivers.
  - For best performance, messages may take different network paths so may arrive out-of-order.
  - Messages from different senders on the same topic can only be ordered when they arrive at the message broker.
- Why can't the message broker reorder messages differently than the arrival order, e.g. using timestamps as keys?
  - This functionality should be provided through a key-value database that is more complicated and much slower.

Communication Challenges for Exactly-Once Delivery

- Network communications are unreliable.
  - Protocols like TCP and HTTP guarantee delivery only when there is no failure.
- Delivery guarantees considering failures
  - At-most once (best effort): messages are sent once, don't use any acknowledgement.
  - At-least once: resend messages until acknowledgements are received.
- Apparently, exactly-once delivery can be achieved by using sequence numbers with at-least once delivery.
  - Message brokers number messages as they arrive.
  - Subscribers and consumers utilize these numbers to acknowledge and reorder messages.
- However, maintaining sequence numbers as messages are generated by publishers and producers is not trivial.
  - Resending unacknowledged messages further complicates the issue as it leads to additional out-of-order arrivals.

Scalability Challenges

- Consider horizontal scalability where multiple message brokers are running on multiple servers.
- Each message broker can handle a number of topics.
  - Similar to the idea of sharding and function partitioning.
- Each topic can be replicated to multiple message brokers.
  - Which then serve huge number of subscribers.
- However,
  - Scaling with multiple publishers and producers sending messages to the same topic is difficult.
  - Scaling for consumers is difficult as replicas need to have consensus on which consumer should process which message.

Outline

Message Queues

Kafka

Apache Kafka

- An open-source distributed event streaming platform.
  - Developed in LinkedIn, open-sourced in 2011
- Features
  - Low-latency message delivery for high volume event streams, e.g. real-time log aggregation and offline data loading.
  - Support a computational model for real-time analytics by consuming and producing event streams.
  - Fault tolerance.

Architecture

- Events as messages are organized and stored in topics.
- A combination of producer-consumer and publisher-subscriber message queues for scalability.
  - Kafka producers publish (write) events to topics.
  - Kafka consumers subscribe to topics and read events within.
  - Events in a topic are partitioned – a group of consumers can read these events in parallel, each for a different partition.
  - Multiple groups of consumers can still read events in a topic as many times as desired.
- Kafka brokers store partitions for topics.
  - Sharding: partitions of a single topic are distributed to multiple brokers for better write performance.
  - Replication: a single partition is replicated across multiple brokers for better read performance, availability, and durability.

Persistence and Performance

- Major factor to drive Kafka design decisions.
- Rely heavily on filesystem for storing and caching messages.
  - Sequential write and read throughput are high enough to saturate network communications.
  - OS automatically makes efficient use of large amount of memory when caching disk data for sequential accesses.
  - On the contrary, most languages cannot use memory as efficiently due to object overhead and garbage collections.
- Disk space is virtually unlimited so messages can be kept for a long time before being deleted.
- To guarantee high performance, only rely on simple and sequential accesses to files.
  - Store messages by appending to files.
  - Serve messages by reading sequentially.
  - Not to support any kinds of indices that would need random accesses – use databases instead.

Message Delivery Semantics

- Assume brokers to work perfectly for now.
- No guarantee among multiple producers within Kafka.
- No ordering among multiple partitions, even if they are from the same topic.
- Exactly-once processing is supported via Kafka transactions.
  - A single transaction reads from and writes to multiple partitions, possibly from different topics.
- There are options for weaker guarantees for other use cases.
  - A single producer with a single partition.
  - A group of consumers with a topic.

Message Delivery for Producers

- A single producer with a single partition.
- At-most once: broker doesn't acknowledge
  - Messages may arrive out of order.
- At-least once: producer resends messages until acknowledged
  - Broker may store a message twice.
  - If a previous message is not acknowledged yet, the next message may arrive out of order.
- Idempotent delivery: producer adds sequence numbers
  - Base on at-least once delivery
  - Broker remove duplicates and acknowledges messages in-order.
  - May affect performance as out-of-order arrivals are not acknowledged and need resend.
  - Only for the lifetime of the producer, no guarantee if it restarts.

Message Processing for Consumers

- A group of consumers with a topic.
- Each consumer reads one partition of the topic.
- Each consumer saves its position within its partition.
  - The position indicates where to read from if the consumer restarts after a failure.
  - However, there are two choices as the consumer need to process the message.
- Process-then-save: at-least once
  - If the consumer fails after processing the message but before saving the position, then when it restarts, it will process the message again.
  - Make sure the processing is idempotent to avoid any issues.
- Save-then-process: at-most once
  - If the consumer fails after saving the position but before processing the message, then when it restarts, it will skip the message.

Replication

- Unit of replication is a partition.
- Consensus determines a leader replica for each partition.
  - Producers write to leader directly.
  - Other replicas (followers) replicate from the leader.
  - Consensus and all states are managed in ZooKeeper.
  - Not P for CAP theorem: no partition tolerance.
- In-sync replicas (ISRs)
  - Replicas that are not too far behind the leader.
  - Messages available from all ISRs are considered committed.
  - Committed messages are less likely to be lost if the leader fails.
  - Consumers only consume committed messages.
  - Producers can choose to receive acknowledgement when the message reaches the leader or when it is committed.

Summary

- Use message queues to decouple message senders and receivers.
- Make a choice between distributed message queues and distributed database systems by considering their performance differences and your application use cases.