Kafka Notes

kafka

L. Brunetti

Kafka è una piattaforma di eventi. Permette di:- pubblicare e sottoscrivere flussi di eventi da diversi sistemi
- salvare eventi per un tempo prestabilito
- processare flussi di eventi direttamente.

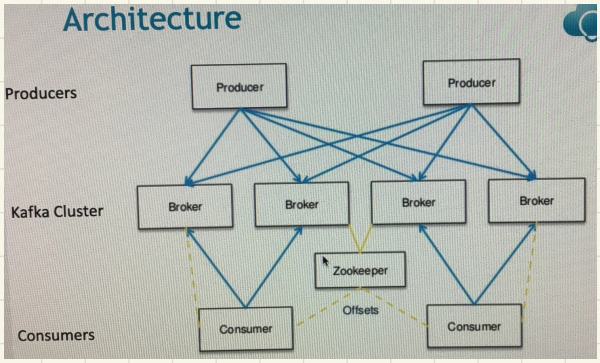
Kafka interviene per disaccoppiare le pipelines, sia di input che di output

Casi d'uso per Kafka → 1. Process Real time 2. Scambi di messaggio 3. Collezionare attività d'utenti
4. Collezionare metriche da applicazioni 5. Log Aggregation 6. Percepire cambio di dati 7. Commit Log sistemi distribuiti

Concepts

Topics: mantiene feeds dei messaggi
Producers: applicazioni o processi che pubblicano sui topic
Consumers: applicazioni o processi che leggono dai topic
Broker: ogni server presente su Kafka.

Tutte le comunicazioni avvengono attraverso protocollo TCP in modo binario
(1. Batching di piccoli messaggi 2. Zero copy I/O usando sendfile)

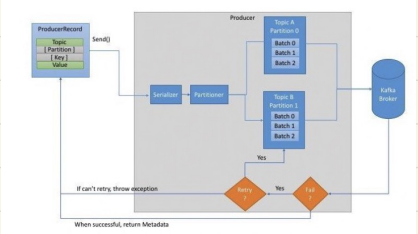


I dati in ogni topic sono partizionati in base ordine di commit, in ogni partizione ad ogni messaggio è associato un ID sequenziale chiamato OFFSET. I dati vengono mantenuti per un periodo di tempo configurabile. I messaggi sono ordinati solo all'interno di una partizione (Meglio avere un solo consumer per Key/partizione).

Producer

-Pubblica sul topic a scelta -Carica in modo distribuito (round robin o partitioning) -Load Balancing dal Broker -Supporta invio async
-Tutti i nodi possono dirci metadata su quali server sono attivi, leader partizione.

Producer Architecture



Producer Configuration

Bootstrap.servers: List of host:port pairs of Kafka brokers. This doesn't have to include all brokers in the cluster, the producer will query these brokers for information about additional brokers. But it is recommended to include at least two

Acks:
- Acks = 0 → Producer will not wait for any reply from the broker before assuming the message was sent successfully. This means that if something went wrong and the broker did not receive the message, the producer will not know about this and the message will be lost
- Acks = 1 → producer will receive a success response from the broker the moment the leader replica received the message. If the message can't be written to the leader (for example, if the leader crashed and a new leader was not elected yet), the Producer will receive an error response and can retry sending the message, avoiding potential loss of data
- Acks = all → Producer will receive a success response from the broker once all in-sync replicas received the message

Producer Configuration

Buffer.memory: the amount of memory the producer will use to buffer messages waiting to be sent to brokers

Compression.type: By default, messages are sent uncompressed. This parameter can be set to snappy or gzip

Retries: will control how many times the producer will retry sending the message before giving up and notifying the client of an issue

Batch.size: the amount of memory in bytes (not messages!) that will be used for each batch

Client.id: just a string for meaningful logs

Max.in.flight.requests.per.connection: how many messages the producer will send to the server without receiving responses. Setting this to 1 will guarantee that messages will be written to the broker in the order they were sent, even when retries occure

Durable Writes

- Producers can choose to trade throughput for durability of writes:

Durability	Behaviour	Per event Latency	Required Acknowledgements (request.required.acks)
Highest	ACK all ISRs have received	Highest	-1
Medium	ACK one the leader has received	Medium	1
Lowest	No ACKs required	Lowest	0

- Throughput can also be raised with **more brokers**
- A sane configuration:

Property	Value
replication	3
min.insync.replicas	2
request.required.acks	-1

Producer Write

Fire-and-forget - in which we send a message to the server and don't really care if it arrived succesfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, some messages will get lost using this method.

Synchronous Send - we send a message, the send() method returns a Future object and we use get() to wait on the future and see if the send() was successful or not

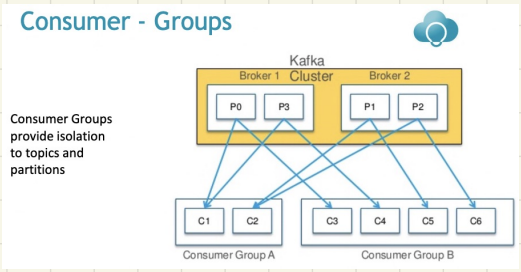
Asynchronous Send - we call the send() method with a callback function, which gets triggered when receive a response from the Kafka broker.

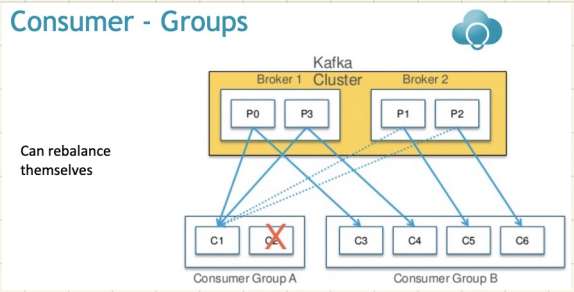
Consumers

- Più consumer possono leggere dallo stesso topic - Ogni Consumer gestisce il suo offset
- I messaggi restano in Kafka, non vengono eliminati dopo essere consumati.

Essi possono essere organizzati in **Consumer Groups**. Esistono due modi:

- **Tutti i consumer in un gruppo:** Agisce come una coda con load balancing
- **Tutti i consumer in gruppi diversi:** i messaggi sono broadcastati a tutti i consumer
- **"Logical Subscriber":** Many consumer in un gruppo: - I consumer sono aggiunti per scalabilità e fault tolerance
  - Ogni consumer legge da uno o più partizioni
  - Non ci possono essere consumer > partizioni





Partitions Rebalance = Evento che avviene quando l'owner di una partizione passa da un consumer all'altro (creazione, rimozione)

Consumer configuration

Enable.auto.commit: then every 5 seconds the consumer will commit the largest offset your client received. With this policy you need to be idempotent.

auto.commit.interval.ms: the interval between commits

Fetch.min.bytes: specify the minimum amount of data that it wants to receive from the broker when fetching records. This reduces the load on both the Consumer and the Broker as they have to handle fewer back-and-forward messages. But it will bring latency spikes.

Fetch.max.wait.ms: By default Kafka will wait up to 500ms trying to match the criteria Fetch.min.bytes.

Session.timeout.ms: The amount of time a consumer can be out of contact with the brokers while still considered alive, defaults to 3 seconds

Consumer configuration

Auto.offset.reset: behavior of the consumer when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid . Latest or earliest

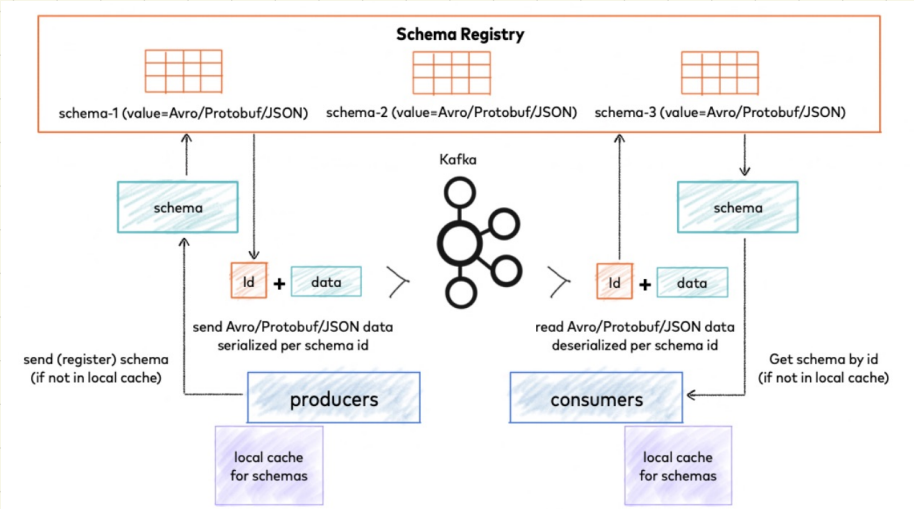
Client.id: just a string for meaningful logs

Confluent Schema Registry

Fornisce un livello di servizio per i metadati, una RESTful interfaccia per memorizzare gli schemi (AVRO, JSON, PROTOBUF).

Gestisce anche uno storico degli schemi in base ad un "Subject Name Strategy" (TopicNameStrategy, RecordNameStrategy, entrambi)

Permette l'evoluzione dello schema fornendo serializers. Schema Registry vive all'interno di Kafka. Producers e Consumers possono riferirsi direttamente a lui per gli schemi



Confluent Schema Registry for storing and retrieving schemas

Schema Registry è un livello di storage distribuito per gli schemi da usare Kafka. Alcuni punti chiave di design sono:

- Assegnare ID univoci ad ogni schema
- Kafka fornisce backend durevole e funge da registro delle modifiche degli schemi
-