Go project

Comet

Overview

Sending asynchronously data across multiple distributed services has always been a non-trivial task for developers. Over the years multiple solutions have paved their way as the leading go-to choices for most use cases - the most notable examples being RabbitMQ, Kafka, HiveMQ, etc.

Core Idea

The goal of this project will be to deliver a similar solution, allowing for different applications to asynchronously communicate between each other. Since there already exist solutions, which tackle this problem, the proposed project will outline the following features:

- Push based feed - clients will keep a constant TCP connection with one of the cluster's brokers and it will forward any incoming events ASAP
- gRPC communication - clients will communicate with the broker based on streaming gRPC calls, which provide a good low level abstraction, giving a good balance between performance, ease of implementation and extensibility
- Almost persistent (Available) - this project is aimed at providing the fastest data stream possible, whilst still maintaining a form of persistence. This in turn means that all messages sent through the system will be processed with ACK=0, i.e. the client sends the event and the broker tries to the best of it's ability to deliver the message to the rest of the brokers. The philosophy is that the incoming data should be so dynamic that missing out on small individual events would not be critical to the user.
- Failure tolerant - if one broker dies out, the client should be able to reach out one of the remaining brokers and still continue to operate with no/minimal data loss.

Domain aggregates

Topics (Metadata)

The core unit of data streaming is the *topic* - a topic represents the place where events can be streamed. The metadata configuration of the different topics will be stored within **etcd** and will cover information about:

- Topic names
- Partitions per topic
- Retention policy - per time, per events or per file size
- Etc.

Consumers

The client services, which will consume the incoming events, are characterized by 2 distinct features:

- Consumer group - this represents the whole given application, e.g. if we have a stream of market data events, one consumer group could be a trading engine, acting upon this data.
- Consumer - this represents an instance, consuming data from zero/one/more topic partitions. In the previous example, if the market data topic has 10 partitions and the trading engine is a single consumer group with 5 consumers, then each consumer will consume 2 unique partitions each.

The data about the current consumer groups and individual consumers will also be stored within **etcd**. There will also be tracking of the latest data which a given consumer has read. This way if for some reason a consumer dies and stops the connection, when it reconnects it will have the opportunity of catching up on lost events.

Events

Once the events have started being sent to the system, they will be stored in a local Write Ahead Log (WAL). In order to keep adequate ordering of the incoming messages, only **partition replication group leaders** can take incoming events. The rest of the brokers within the partition replication group can act only as read-only replicas.

The partition replica leader will keep an active gRPC connection to the rest of the brokers within the group. As brokers receive messages between each other they will propagate them to the clients. **There will be no advanced synchronization between brokers** - the data will be sent in a “fire and forget” manner, which means that if a broker goes down for some time, its WAL can be out of sync.

Client-side

Since this is a streaming platform, there will be a need to have a client library, which will keep track of the following:

- Function callbacks - when the client receives an event it must know to which user handler it must forward it for processing.
- Cluster health - if the currently active broker suddenly dies out, the client library must be able to automatically reroute to the next healthy broker.
- Data formatting - the library will be responsible for keeping track of the annotated user types and thus converting the incoming event data from raw bytes into the user-defined types.

Demo

In order to demonstrate this project, I’ll create a very simple project where one service will read incoming quotes/LTPs (Last Trade Prices) from a given market data provider (e.g.

Alpaca, CoinDesk, etc.) and will stream the topics via Comet. Then there will be a second service, which will act as a consumer for the prices.

Endpoints

gRPC

The primary data will be sent over gRPC, so the main methods will be:

- `rpc FetchMetadata() returns (ClusterMetadataResponse)`
- `rpc StreamEvents() returns (stream EvenMessage)`

The `StreamEvents()` procedure can work both ways - from the broker to the client and from the client to the broker, so there will be no need for adding in extra layers of complexity.