1-Pager: Testing Strategy for New Kafka Consumer

This is a follow up task for the:Consumer threading refactor design

Objective

To ensure that our new Kafka consumer is robust, performant, and scalable by rigorously testing its capabilities in various scenarios, including different message sizes, numbers, consumer counts, and CPU throttling conditions.

Scope

1. **Consumption Rate**
    - Varying message sizes: 100B, 1KB, 10KB, 100KB
    - Varying message numbers: 100, 1000, 10,000, 100,000
2. **CPU Throttling**
    - No throttling
    - 99.99%
3. **Special Scenarios**
    - Schema registry usage
    - Slow schema resolution

Performance Testing Strategy

Setup

- Kafka Cluster: n-broker setup (do we need more than 1?)
- Producer: Pre-configured to produce varying sizes and volumes of messages
- Consumer: Async Consumer and the current KafkaConsumer implementation

Consumption Rate

1. **Varying Message Sizes**: Measure the rate of message consumption across different message sizes.
    - Metrics: Throughput (messages/sec), Latency
    - Tools: Kafka built-in monitoring, custom logging
2. **Varying Message Numbers**: Measure how well the consumer handles varying amounts of messages.
    - Metrics: Throughput, Backlog drain time
    - Tools: Kafka monitoring, custom logging**CPU Throttling**

1. **No Throttling**: Baseline performance metrics.
    - Metrics: Throughput, CPU, and Memory Usage
2. **50% and 75% Throttling**: Simulate CPU constraint scenarios.
    - Metrics: Throughput, Latency, CPU and Memory Usage

Special Scenarios

1. **High Deserialization CPU Cost**: Simulate a high-CPU cost deserialization algorithm.
    - Metrics: Throughput, Latency, CPU Utilization
    - Tools: Kafka monitoring, Profiling tools
2. **Schema Registry**: Measure the impact of using a schema registry for deserialization.
    - Metrics: Throughput, Latency, Schema registry lookup time
    - Tools: Kafka monitoring, Schema Registry logs

Stochastic Testing Strategy

Goal

We need to deterministically emulate all possible real-world usage of the async consumer. Despite integration testing and unit testing cover some aspect of it, I think it is necessary for us to try to generate a large number of usage patterns and verify the consequence of these actions. For example - offsetComit follows by consumer.subscribe should yield nothing because the consumer has not made any progress. Consumer poll should trigger auto commit and we need to verify that the previously return data was committed to the coordinator.

The goal is to simulate a series of events in pseudo-randomly and verify the outcome of each action. If an unexpected result is detected, we should be able to retrieve the sequence of actions so that we can debug the issue.

Components

A random action/sequence generator: We should generate the next action based on the current state of the consumer by probability.

State tracker: We record the state of the consumer and predict the consequence of the following action.

A sequence logger: The actions are logged into a sequence of events, chronologically so that we can always reproduce the actions.

Response simulator: Based on the request sent from the client, we pseudo-randomlly generate responses for a given request. Each response can be branched into a separate consumer state for testing.