Data Mover Go application NFR - Non Functional Requirements Q&A.

WA Brown aes@3pp.com

Is Data-Mover multi-threaded?

Yes, **Data-Mover** is multi-threaded. Both the Pub/Sub subscriber and Kafka producer inherently handle concurrent operations:

1. **Pub/Sub Subscriber**:
   - The Subscription.Receive method spawns goroutines to invoke the message handler concurrently for each message.
   - The concurrency is managed by the MaxOutstandingMessages parameter in ReceiveSettings.
2. **Kafka Producer**:
   - The Kafka producer uses an internal thread pool for sending messages asynchronously.
   - Delivery reports are processed in a separate goroutine created within the NewProducer function.

Will it handle concurrency? How? What number of concurrent calls?

Yes, **Data-Mover** handles concurrency effectively:

1. **Pub/Sub**:
   - The Subscription.Receive function supports concurrent message processing by default.
   - The concurrency level is determined by the MaxOutstandingMessages setting. In this code, it is set to 100, meaning up to 100 messages can be processed concurrently.
2. **Kafka**:
   - Kafka producer supports high concurrency internally by allowing multiple threads to publish messages asynchronously. No explicit concurrency limits are defined in the producer.

Number of Concurrent Calls:

- **Pub/Sub**: Up to 100 concurrent messages (as configured).
- **Kafka**: Concurrency depends on Kafka’s internal configuration and broker capacity, which can handle thousands of messages concurrently if configured properly.

Will it scale? How? How many jobs per second?

Yes, **Data-Mover** will scale effectively due to its architecture:

1. **Scaling Mechanisms**:
   - **Pub/Sub**: Google Pub/Sub can scale dynamically to handle a large

    number of messages by increasing the number of subscribers or their MaxOutstandingMessages limits.
    - **Kafka**: Kafka is inherently scalable by adding partitions to topics and increasing the number of brokers in the cluster.
2. **Horizontal Scaling**:
    - The application can scale horizontally by deploying multiple instances of **Data-Mover**, each with its own subscriber and producer.
3. **Jobs Per Second**:
    - Pub/Sub supports millions of messages per second, so the limit is determined by:
        - The number of subscribers (MaxOutstandingMessages).
        - The speed of processing and network throughput.
        - Kafka broker capacity.
4. **Estimate**: With current settings (100 concurrent messages and assuming minimal processing delay), **Data-Mover** could handle hundreds of messages per second per instance, assuming no bottlenecks in Kafka or Pub/Sub.

Is the current error handling sufficient? If not, what should be added?

Current Error Handling:

1. **Pub/Sub Subscriber**:
    - Acks or Nacks messages based on the success of the MessageHandler.
    - Logs errors when messages fail to process.
2. **Kafka Producer**:
    - Logs errors for message delivery failures using Kafka's delivery report mechanism.
3. **Bridge**:
    - Updates failure metrics for both Pub/Sub and Kafka errors.
    - Logs errors for failed message processing.

Improvements:

1. **Retry Mechanism**:
    - Add a retry policy for failed messages (both Pub/Sub and Kafka).
    - Pub/Sub messages are Nacked, which allows retries by Pub/Sub, but Kafka message failures should include a retry mechanism before logging as a failure.
2. **Dead Letter Queue (DLQ)**:
    - Implement a DLQ for messages that cannot be processed after multiple retries.
    - Pub/Sub supports DLQs natively, while Kafka DLQs can be

implemented as a separate topic.

3. **Circuit Breaker Pattern**:
   - Prevent cascading failures during high error rates by pausing processing temporarily.
4. **Backpressure Handling**:
   - Monitor Kafka producer queue size and slow down Pub/Sub message intake if the Kafka producer cannot keep up.
5. **Metrics Alerts**:
   - Add alerting mechanisms for key metrics (e.g., high failure rates, increased processing time).

Summary:

1. **Concurrency**: Data-Mover supports concurrent message processing with both Pub/Sub and Kafka.
2. **Scalability**: It can scale horizontally and handle potentially thousands of jobs per second, limited by resource configuration.
3. **Error Handling**: Current handling is basic but functional. Adding retries, DLQs, circuit breakers, and alerting would make it robust.