CS60002 Distributed Systems

Assignment 1: Implementing a Distributed Queue

Date: Jan 16th, 2023

Target Deadline (For sharing the git repo): Jan 31st, 2023

Note that there will be NO extension to this deadline

In this assignment, you will get familiar with the distributed queue and learn how to implement a distributed logging queue.

Submission Instructions:

Please submit your code, along with any necessary documentation on Github. Note that some parts of this assignment may need your own design choices, so feel free to design your solution accordingly. However, you **must** include a brief write-up (1-2 pages) explaining your design decisions and the testing that you conducted to ensure the correctness and efficiency of your implementation. This should include:

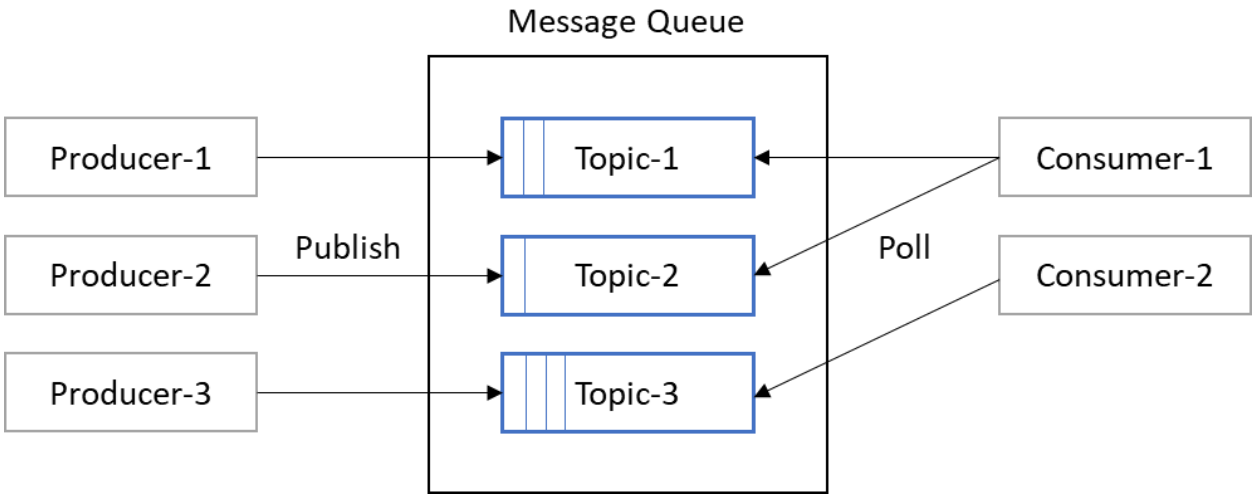
I. A description of your implementation approach and any trade-offs you considered.
II. A description of the testing you conducted, including the types of tests you ran and the results you obtained.
III. A discussion of any challenges or difficulties you encountered during the implementation and testing process.
IV. You should clearly mention any hyperparameters that you assume during your implementation.

Grading Scheme:

- Correctness: 50%
- Report: 15%
- Presentation: 20%
- Code quality: 15%

Introduction:

As a software engineer at **Connectify**, a new social media company, you have been tasked with implementing a distributed logging queue to store log messages generated by the company's various applications and services.



The distributed queue should be able to handle multiple producers and consumers that can subscribe to different topics. Different producers can publish topics asynchronously over the queue, whereas multiple consumers can retrieve the topics asynchronously, for which they are subscribed to.

A **topic** is a label that categorizes log messages in a distributed queue. Topics can be used to group log messages related to each other or to allow consumers to subscribe to specific categories of log messages.

Producers should be able to add log messages to the queue with a specified topic. Consumers should be able to retrieve log messages from the queue based on the topics they are subscribed to.

For example, the distributed queue at Connectify might have the following topics:

- "post_created" for log messages related to new posts being created on the platform
- "user_signup" for log messages related to new users signing up for the platform
- "user_login" for log messages related to users logging into the platform

Part-A

Design and implement a distributed logging queue using a programming language of your choice. Your broker implementation should support the following HTTP APIs:

a. CreateTopic

This operation allows you to create a new topic in the queue. The topic would be added to a list of available topics, and producers and consumers would be able to subscribe to it. This operation takes a string parameter specifying the name of the topic to be created.

```
Method: POST
Endpoint: /topics
Params:
    - "name": <string>
Response:
    onSuccess:
        - "status": “success”
        - "message": <string>
    onFailure:
        - "status": “failure”
        - "message": <string>  // informative error message
```

An example is given below for your reference:
Request Body:

```
{
  "topic_name": "example_topic"
}
```

Response Body:

```
{
    "status": "success",
    "message": "Topic 'example_topic' created successfully"
}
```

Response Code:

```
200 OK
```

b. ListTopics

This operation allows you to retrieve a list of all available topics in the distributed queue.

```
Method: GET
Endpoint: /topics
Params:
    None
Response:
    "status": <string>
    onSuccess:
        - "topics": List[<string>]  // List of topic names
    onFailure:
        - "message": <string>  // Error message
```

c. RegisterConsumer

This operation allows a consumer to register for a specific topic with the queue. If the consumer is interested in more than one topic, they call this endpoint multiple times, each time getting a topic specific consumer_id. This endpoint should return an error if the requested topic doesn't exist.

```
Method: POST
Endpoint: /consumer/register
Params:
    "topic": <string>
Response:
    "status": <string>
    onSuccess:
```

```
        - "consumer_id": <int>
    onFailure:
        - "message": <string>  // Error message
```

d. RegisterProducer

This operation allows a producer to register for a specific topic with the queue. A producer should register with exactly one topic. If the requested topic doesn't exist, calling this endpoint should create the topic.

```
Method: POST
Endpoint: /producer/register
Params:
    "topic": <string>
Response:
    "status": <string>
    onSuccess:
        - "producer_id": <int>
    onFailure:
        - "message": <string>  // Error message
```

e. Enqueue

Add a log message to the queue.

```
Method: POST
Endpoint: /producer/produce
Params:
    - "topic": <string>
    - "producer_id": <int>
    - "message": <string>  // Log message to enqueue
Response:
    "status": <string>
    onSuccess:
        None
    onFailure:
        - "message": <string>  // Error message
```

f. Dequeue

Remove and return the next log message in the queue for the specified topic for this consumer.

```
Method: GET
Endpoint: /consumer/consume
Params:
    - "topic": <string>
    - "consumer_id": <int>
Response:
    "status": <string>
    onSuccess:
        - "message": <string>  // Log message
    onFailure:
        - "message": <string>  // Error message
```

g. Size

Return the number of log messages in the requested topic for this consumer.

```
Method: GET
Endpoint: /size
Params:
    - "topic": <string>
    - "consumer_id": <int>
Response:
    "status": <string>
    onSuccess:
        - "size": <int>
    onFailure:
        - "message": <string>  // Error message
```

Part-B (Add persistence to the distributed queue)

The current implementation of the distributed queue stores log messages in memory, which means that the queue is lost if the server crashes or restarts. This is a problem because it means that important log messages might be lost, and the queue cannot be restored to a previous state if necessary.

Your task is to add persistence to the distributed queue, so that log messages are stored in a persistent storage layer (use a sql storage, preferably postgres) and can be retrieved even if the server crashes or restarts. The queue should be able to recover from failures and continue functioning as normal.

To solve this problem, you will need to:

- Determine a suitable schema for persistent storage layer for the log messages
    - For example you can have the following tables `Topic`, `Producer`, `Consumer`, `Log`, you can decide other meta fields besides the id, name, and other required data fields like the entries of the Log for a topic.
- Modify the queue implementation from part A to store and retrieve log messages from the persistent storage layer
- Test the queue to ensure that it is able to recover from failures and continue functioning as expected

Part-C (Implement client libraries for producer and consumer clients.)

The current implementation of the distributed queue uses HTTP requests to communicate with producers and consumers. However, it can be inconvenient or inefficient for clients to directly call these endpoints.

Your task is to implement client libraries in a programming language of your choice (e.g. Java, Go, Python, etc.) that allow producers and consumers to communicate with the queue using a more convenient or efficient interface. The client libraries should call the existing HTTP endpoints under the hood, but provide a higher-level interface to the clients.

A client library, also known as a software development kit (SDK), is a set of tools and libraries that developers can use to interact with a specific software or service. It provides an easy-to-use interface for developers to access the functionality of the software or service, allowing them to build applications and integrations without having to understand the underlying implementation details.

You need to implement a package/library, an example use of myqueue library with MyProducer and MyConsumer classes is provided below.

Producer client:

```
from myqueue import MyProducer

def produce():
    producer = MyProducer(
        topics=['topic1', 'topic2'],
        broker='localhost:9092')
    try:
        while producer.can_send():
            # Produce message and wait for it to be sent
            producer.send("my_topic", b"DEBUG: This is a log
message!")
    finally:
        # Wait for all messages to be delivered or expire
        producer.stop()
```

Consumer client:

```
from myqueue import MyConsumer

def consume():
    consumer = MyConsumer(
       topics=[ 'topic1', 'topic2'],
       broker='localhost:9092')
    try:
        # Consume messages
        for msg in consumer.get_next():
            # get_next() can be implemented as a generator
            print("consumed: ", msg.topic)
    finally:
        consumer.stop()
```