Code Documentation

Group Number- 12

Github Link: https://github.com/AgrawalDrishti/SDE_MAJOR_PROJECT

1. ***broker*** ***:*** Registers the Broker Service and Manages Topic-Based Message Operations

Parameters :

- **req (Object):**
  Incoming HTTP request object used to extract parameters (e.g., topic, messages).
- **res (Object):**
  HTTP response object used to send back the result of operations.
- **socket (Socket):**
  WebSocket connection object used for real-time communication.

Process Flow

1. **Server Initialization**
   - Set up an HTTP server and WebSocket (`socket.io`) server.
   - Reads configuration from environment variables and arguments.
   - Initializes *`MESSAGE_DIRECTORY`* to store broker message files.

2. **Socket Event Handlers**
   - Client Connection/Disconnection:
     Logs when a client connects or disconnects.
   - `publish` Event:
     - Publishes a message to a topic.
     - Saves the message in a file corresponding to the topic.

    - Updates the `MessagesToUpdate` queue for real-time message propagation.
- `consumeTopic` Event:
    - Allows clients to consume messages from a topic starting from a specified offset.
    - Reads the topic's message file and returns the messages.

3. **REST API Endpoints**
    - **GET `/`**: Returns a health status message of the broker.
    - **GET `/topics`**: Responds with a list of all available topics.
    - **POST `/addTopic`**:
        - Adds a new topic to the broker.
        - Creates a corresponding message file.
        - Error handling ensures proper file creation.
    - **POST `/updateTopicMessages`**:
        - Updates a topic's messages based on incoming requests.
        - Verifies the presence of topic and messages in the request body.
    - **GET `/followers`**: Retrieves the follower brokers mapped to each topic.
    - **POST `/setFollowers`**:
        - Maps follower brokers to a topic.
        - Validates the request body for both topic and followers.
    - **POST `/addFollower`**:
        - Adds a single follower broker to a topic.
        - Initializes the follower list if not already present.

4. Broker Registration with Zookeeper

- Registers the broker with the Zookeeper system via an HTTP POST request.
- Handles errors during registration gracefully and shuts down if necessary.

5. Message Synchronization

- Periodically propagates pending messages *(`MessagesToUpdate`)* to follower brokers.
- Uses an *`async-mutex`* to handle concurrency while accessing shared resources.

2. *file_handler:* The *file_handler* module provides utility functions for handling file and directory operations. These include creating files and directories, reading file content from specific positions, and appending messages to existing files. It simplifies file management tasks with robust error handling and logging mechanisms to ensure smooth execution.

a. create_message_file

Creates a file with specified content in a given directory.

Parameters

- dir (String): Directory path where the file should be created.
- fileName (String): Name of the file to be created.
- content (String, optional): Content to be written to the file. Defaults to an empty string.

Process Flow

1. Constructs the file path using the directory and file name.
2. Creates the file with the provided content using fs.writeFile.
3. Logs a success message or throws an error on failure.

b. read_message_file

Reads a file starting from a specified position, extracting content until the next newline or buffer limit.

Parameters

- dir (String): Directory path where the file is located.

- fileName (String): Name of the file to be read.
- startpos (Number): Position in the file to start reading from.

Process Flow

1. Constructs the file path using the directory and file name.
2. Opens the file in read mode and reads up to 1024 bytes from the start position.
3. Extracts content up to the next newline or the end of the buffer.
4. Closes the file descriptor after reading.

c. create_message_directory

Creates a new directory if it does not already exist.

Parameters

- dir (String): Path of the directory to be created.

Process Flow

1. Checks if the directory exists.
2. Creates the directory if it does not exist using fs.mkdirSync.
3. Logs whether the directory was created or already exists.

d. add_message_to_file

Appends a message to an existing file, followed by a newline.

Parameters

- dir (String): Directory path where the file is located.
- fileName (String): Name of the file to append content to.
- content (String): The content to append.

Process Flow

1. Constructs the file path.
2. Appends the provided content followed by a newline using fs.appendFile.
3. Logs a success message or throws an error on failure.

3. ***consumer:*** The *consumer* module provides functionality to consume data streams from specified topics in real time. It interacts with brokers using WebSocket connections and efficiently handles reconnections and message offsets, ensuring reliable data retrieval.

a. startConsuming

Begins consuming messages from a specified topic using a WebSocket connection to the appropriate broker.

Parameters

- topic (String): The topic name from which messages are to be consumed.

Process Flow

1. Initialize Socket Connection:
    a. Retrieves the broker URL from the mapping object based on the topic.
    b. Establishes a WebSocket connection using socket.io.
    c. Ensures the consumedLength for the topic is initialized to track the message offset.
2. Periodic Consumption:
    a. Sets up a 5-second interval to emit a consumeTopic event to the broker.
    b. Passes the topic and current consumedLength to request new messages.
    c. Processes the broker's response:
        - Logs the messages to the console.
        - Updates the *consumedLength* based on the length of the consumed data.
        - Handles unexpected response formats with error logging.
3. Handle Disconnection:
    a. Listens for the disconnect event from the WebSocket.

   b. Logs the disconnection and retrieves a new broker URL for the topic from Zookeeper.
   c. Updates the mapping and restarts the consumption process.

Returns

1. Continuously logs the consumed messages and updates the offset for each topic.
2. Reconnects and resumes consumption automatically if the connection to the broker is lost.

4. ***producer:*** The *producer* module provides functionality for publishing messages to specific topics in real time. It establishes a WebSocket connection with brokers, handles disconnections gracefully, and ensures seamless message publication using a periodic mechanism.

a. startPublishing

Begins publishing messages to a specified topic using a WebSocket connection to the appropriate broker.

Parameters

- topic (String): The topic name to which messages are to be published.

Process Flow

1. Initialize Socket Connection:
   a. Retrieves the broker URL from the *mapping* object based on the topic.
   b. Establishes a WebSocket connection using *socket.io*.
2. Periodic Message Publishing:
   a. Sets up a 2-second interval to emit a *publish* event to the broker.

   b. The message contains the topic name and a unique identifier (*topic+i*).
   c. Uses a callback to log errors or broker responses through a *logger* function.
   d. Increments the identifier (*i*) with each published message.
3. Handle Disconnection:
   a. Listens for the *disconnect* event from the WebSocket.
   b. Logs the disconnection and attempts to retrieve a new broker URL for the topic by sending a *ping* request to Zookeeper.
   c. Updates the *mapping* with the new broker URL and restarts the publishing process.
   d. Logs errors if a new broker cannot be found.

Returns

1. Continuously publishes messages to the specified topic.
2. Automatically reconnects and resumes publishing if the connection to the broker is lost.

5. ***topic_controller:*** The *topic_controller*module provides functions to manage topics across brokers, including adding topics with replication and removing topics from the system. It ensures topic consistency and replication using leader-follower relationships and handles concurrency with a mutex lock.

a. add_topic_to_broker

Adds a new topic to the broker system, assigning a leader broker and replicating the topic across follower brokers.

Parameters

- req (Object): The request object containing the topic to be added (*req.body.topic*).

- res (Object): The response object used to send back the operation's result.

Process Flow

1. Mutex Acquisition:
    a. Acquires a mutex lock to ensure that only one topic addition process runs at a time.
2. Assign Leader Broker:
    a. Selects the current broker (*brokers[broker_i]*) as the leader for the topic.
    b. Sends a POST request to the leader broker to add the topic.
3. Set Follower Brokers:
    a. On successful leader assignment, selects a random sample of brokers (excluding the leader) as followers.
    b. Sends a POST request to the leader broker to register these followers for the topic.
    c. Adds the topic to each follower broker using individual POST requests.
4. Update Broker Mappings:
    a. Updates *TopicLeaderBrokerMap* with the leader broker for the topic.
    b. Updates *TopicFollowerBrokersMap* with the follower brokers for the topic.
    c. Advances the round-robin pointer (*broker_i*) for the next topic assignment.
5. Error Handling:
    a. Logs and sends appropriate error messages for failures during leader or follower addition.
6. Mutex Release:
    a. Releases the mutex lock after completing the operation.

Returns

1. *200 OK*: If the topic and its followers are successfully added.
2. *500 Internal Server Error*: If an error occurs during the process.

b. remove_topic_from_broker

Removes a topic from the leader broker mapping and the system.

Parameters

- req (Object): The request object containing the topic to be removed(*req.body.topic*).
- res (Object): The response object used to send back the operation's result.

Process Flow

1. Check Topic Existence:
    a. Verifies if the topic exists in *TopicLeaderBrokerMap*.
2. Remove Topic:
    a. Deletes the topic from *TopicLeaderBrokerMap*.
    b. Logs a success message.
3. Error Handling:
    a. Returns a *404 Not Found* response if the topic does not exist.
    b. Returns a *500 Internal Server* Error response for unexpected errors.

Returns

1. *200 OK:* If the topic is successfully removed.
2. *404 Not Found:* If the topic is not found in the system.
3. *500 Internal Server Error:* If an error occurs during the process.

6. ***broker_controller:*** The *broker_manager* module manages brokers in the system, enabling registration, monitoring of their health, and leader election in case of failure. It ensures consistent broker assignments and smooth operation by handling dynamic changes in broker status.

a. add_broker

Registers a new broker in the system and updates the broker list maintained by Zookeeper.

Parameters

- req (Object): The request object containing the broker URL (*req.body.broker_url*).
- res (Object): The response object used to send back the operation's result.

Process Flow

1. Retrieve Broker URL:
    a. Extracts the broker URL from the request body.
2. Add Broker:
    a. Logs the broker addition process.
    b. Adds the broker URL to the brokers list.
3. Error Handling:
    a. Sends a 400 Bad Request response if an error occurs.

Returns

1. *200 OK*: If the broker is successfully added.
2. *400 Bad Request*: If an error occurs during broker registration.

b. ping_broker

Monitors the health of a broker and performs leader election if the broker becomes unresponsive.

Parameters

- req (Object): The request object containing the topic and broker URL (*req.body.topic* and *req.body.broker_url*).
- res (Object): The response object used to send back the operation's result.

Process Flow

1. Check Broker Health:
    a. Sends a *GET* request to the leader broker for the topic.
    b. Logs whether the broker is alive or busy based on its response status.
2. Handle Unresponsive Broker:
    a. If the broker is unresponsive:
        - Logs the broker failure.
        - Initiates leader election by assigning a new leader broker from the topic's followers.
        - Updates *TopicLeaderBrokerMap* with the new leader.
        - Updates the *brokers* list by removing the unresponsive broker.
        - Sends a *POST* request to the new leader to register its followers.
3. Error Handling:
    a. Logs the status of the leader election process and updates mappings.

Returns

1. *200 OK*: If the broker is healthy, busy, or a new leader is successfully elected.
2. *500 Internal Server Error*: If an error occurs during broker monitoring.

7. ***zookeeper:*** The *zookeeper* module acts as the central coordination service for a distributed system. It provides APIs for managing brokers and topics, monitoring their status, and maintaining leader-follower mappings to ensure replication and fault tolerance.

   Purpose:

   Orchestrates topic and broker management within the system by maintaining mappings of leaders and followers, handling broker addition, topic replication, and leader election in case of broker failures.

   Global Variables:

   - *REPLICATION_FACTOR*: Number of replicas per topic, defaults to 2.
   - *TopicLeaderBrokerMap*: Stores leader broker for each topic.
   - *TopicFollowerBrokersMap*: Stores follower brokers for each topic.
   - *mutex*: Ensures concurrency control during critical operations.
   - *broker_i*: Keeps track of the current broker in a round-robin manner.
   - *brokers*: List of registered brokers in the system.

   External Dependencies:

   - *dotenv*: Loads environment variables.
   - *express*: Sets up RESTful APIs.
   - *async-mutex*: Provides a mutex for concurrency control.
   - *cors*: Enables cross-origin requests.
   - *axios*: Facilitates HTTP requests for internal and external communication.

APIs defined in zookeeper:

a. **GET /**
   Checks the health of the Zookeeper service.
   - Response:
     200 OK: "Zookeeper running at *ZOOKEEPER_PORT*"

b. **GET /getMapping**
   Retrieves the leader broker mapping for all topics.
   - Response:
     200 OK: *TopicLeaderBrokerMap*

c. **GET /getFollowers**
   Retrieves the follower brokers mapping for all topics.
   - Response:
     200 OK: *TopicFollowerBrokersMap*

d. **Broker Management APIs**
   - *GET /broker/all*: Retrieve all brokers.
   - *POST /broker/add*: Add a new broker.
   - *POST /broker/ping*: Monitor broker health and perform leader election if necessary.

e. **Topic Management APIs**
   - *POST /topic/add*: Add a topic to the system and replicate it.
   - *DELETE /topic/remove*: Remove a topic from the system.

Structure

a. **Controllers:**
   - ■ **Broker Controller** (*broker_controller*): Manages broker registration and health checks.

- **Topic Controller** (*topic_controller*): Handles topic addition, replication, and removal.

b. **Routers:**
  - **Broker Routes** (*broker_routes*):
    - Handles broker-specific APIs.
  - **Topic Routes** (*topic_routes*):
    - Handles topic-specific APIs.

c. **Global Initialization**:
  - Initializes essential variables, including replication factor and broker/topic mappings.

d. **Service Host**:
  - Starts the Zookeeper service on the configured host and port, logging the initialization status.