Lab 3 Design Document - RAFT Consensus algorithm

order.py

Overview:
The code defines a HTTP server that serves requests for placing orders for toys. The server receives POST requests, extracts the parameters for the order, validates the request parameters, and then sends a request to another catalog service to check for stock availability, and places the order if the quantity is enough. The program is written in Python, and the HTTP server is implemented using the http.server module. Here, we also implement Replication and Raft algorithm.

Design Choices:
The program has a single class OrderRequestHandler which inherits from the http.server.BaseHTTPRequestHandler class. The class OrderRequestHandler handles the POST requests, validates the order parameters, and forwards the request to another catalog service for checking the availability of the toy and its stock. The server is multithreaded to allow multiple clients to connect to the server concurrently. The multithreading is implemented using the ThreadingMixIn class provided by the socketserver module. The ThreadPoolExecutor class is used to manage a pool of threads for executing incoming requests. It creates a pool of worker threads with a maximum number of workers set to the number of CPU cores (multiprocessing.cpu_count()). When a new request is received, the executor assigns it to an available thread from the pool, allowing multiple requests to be processed concurrently. The program uses a read-write lock to ensure thread safety when accessing the catalog service.

OrderRequestHandler class:
The OrderRequestHandler class handles incoming requests and sends the requests to the catalog service. The code handles ‘order’ requests from a client and forwards them to the catalog service. It first checks if the requested toy is present in the catalog service then it validates that catalog service has enough stocks available by the order function for buy by making a request to it and retrieving the remaining quantity of the specified stock. If there are enough stocks available, it then places the order by sending a POST request to the catalog service with the required parameters such as toy name and quantity. It then checks if the order is successful by verifying the response status code, and if not, raises a runtime error. It then updates the order log with the order number, toy name, quantity, and writes the order log into orders.csv. If the order is successful, it sends a response to the client with the order number and a CSV file containing the order log. However, if there is not enough remaining quantity or toy not present in the catalog file, it sends an error response to the client with an appropriate error message. The class is responsible for validating the request parameters and returning an appropriate response to the client.

The max_workers count in ThreadPoolExecutor is set to the number of available CPU cores to efficiently utilize system resources.

Raft class:

The Raft consensus algorithm is designed to ensure fault-tolerance and consistency in distributed systems. This document outlines the implementation details of the Raft class, focusing on key functionalities such as initializing log files, managing peer addresses, replicating logs, and handling leader and follower crashes.

Key Functionality Details:

Attributes:

The __init__ method is crucial as it sets up the initial state of the Raft instance. Here are the key attributes:

log_file: Represents the path to the log file where Raft logs are stored.
follower_addresses: Holds the addresses of follower nodes for communication.
current_term: Tracks the current term in the Raft algorithm.
failed_follower: Keeps a list of followers that have failed or encountered issues.
rwlock: Manages the locking mechanism for thread-safe operations.
index: Acts as an auto-incrementing index for log entries, ensuring each log has a unique identifier.

Methods:

1. init_logfile(): This method initializes the log file if it doesn't exist. It creates a TXT file with specific field names ('Index', 'Timestamp', 'Term', 'Event Type', 'Details') separated by '|' as a delimiter. This setup prepares the log file structure for storing Raft log entries.
2. append_log_entry(term, event_type, details): This method appends a new log entry to the log file. It takes parameters such as the term, event type, and details of the log entry, generates a timestamp, increments the index for each entry, and writes these values to the log file in a TXT format.
3. init_peeraddr(): This method initializes peer addresses based on environment variables. It assumes a specific naming convention for these variables (ORDER_HOST_{follower_id} and ORDER_PORT_{follower_id}) to fetch follower node addresses. These addresses are then stored in the follower_addresses attribute for later use in communication.
4. replicate_logs(log_entries, follower_addresses): This method replicates log entries to follower nodes. It takes log entries in JSON format and a list of follower addresses as parameters. Using HTTP POST requests, it sends the log entries to each follower node through /followers-log API , and checks for successful replication. In case of failures, it tracks failed follower nodes.
5. get_lastterm(): This method simply returns the current term in the Raft algorithm. It provides an interface to access the current term value from outside the class.
6. get_lastindex(): Returns the last index of the log file.
6. fetch_replica_last_log(): Function retrieves the index number of the last log entry from a CSV log file, returning 0 if the file is empty or doesn't exist.
7. Fetch_log_entry_by_index(): Function retrieves the data from TXT file for particular index value. This function is considered when replica synchronization is required i.e. replica's last log index doesn't match up with leader's last index.

The placeholder method (handle_leader_crash()) is not implemented in the provided code but is a placeholder for future implementation. They would handle scenarios like leader crashes , including leader re-election logic and ensuring log consistency in the cluster.

Design Choice:

The Order Server provides a set of APIs for managing toy orders and coordinating with other services in the system. This document outlines the design of these APIs, including their endpoints, request/response formats, and functionality.

API Endpoints

1. POST /orders

- Description: Place a new toy order.
- Request Body:
  name: Toy name (string)
  quantity: Quantity of the toy (integer)
- Response:
  200 OK: Order placed successfully. Returns order number.
  400 Bad Request: Invalid request format or quantity is zero.
  503 Service Unavailable: Failed to replicate order to follower nodes.
  Raise exception: Invalid URL or network error

2. POST /leaderselection

- Description: Notify the server about leader selection.
- Request Body:
  leader_id: ID of the leader node (integer)
  message: Leader selection message ("You win")
- Response:
  200 OK: Leader selection acknowledged.
  Bad Request: Invalid request format. Raise exception

3. POST /inform_replica

- Description: Inform replicas about the current leader.
- Request Body:
  leader_id: ID of the leader node (integer)
- Response:
  200 OK: Replica informed about the leader.
  Bad Request: Raise exception

4. POST /followers-log

- Description: Follower nodes receive log entries from leader.
- Request Body:
  term: Current term (integer)
  event_type: Type of event ("ORDER REQUESTED" or other events)
  details: Event details (string)
- Response:

200 OK: Log entry received successfully.
400 Bad Request: Invalid JSON format.

5. POST /deletelastline

- Utilized to remove the last line from a log file in scenarios where errors occur during log replication or processing i.e if the leader doesn't receive any response from more than half of the the follower nodes that they have appended the entries in respective log. This ensures log integrity and error handling in distributed systems. It's essential for maintaining accurate and reliable logging functionality.
- Get the minimum line number from all the log files.
- Request Body:
  line_number: Minimum line number to keep (integer)
- Response:
  200 OK: Last line deleted successfully.
  Raise RequestException

6. POST /replicate_data_nodes

- Description: Replicate order data to follower nodes.
- Request Body:
  order_number: Order number (integer)
  name: Toy name (string)
  quantity: Quantity of the toy (integer)
- Response:
  200 OK: Order data replicated successfully.

7. GET/get_last_log_entry

- Description: Facilitates fetching the log entry at a specified index. Upon receiving a request with the log index, it queries the server's Raft instance to fetch the corresponding log entry. If the entry is found, it responds with a success status code (200) along with the log entry in JSON format; otherwise, it responds with a failure status code (400).

Functionality Overview

Order Placement:

The **/orders** endpoint allows clients to place toy orders. Validations are performed on the request data, and orders are logged and replicated to follower nodes.

Leader Selection and Notification:

Leader selection and notification APIs (**/leaderselection** and **/inform_replica**) handle leader election events and inform replicas about the current leader.

Log Management:

APIs like **/followers-log** receive log entries from follower nodes, ensuring log consistency and replication across the system.

File Operations:

The **/deletelastline** endpoint supports file operations, specifically deleting the last line from a log file of the follower node when the leader undone the request.

Data Replication:

The **/replicate_data_nodes** API replicates order data to follower nodes, ensuring data consistency and availability.

Replicas crash

In the event of a replica failure, the synchronization process initiates by comparing the current log index of the replicas with the last index in the leader's log file. This synchronization repeats until the replica's last log index matches the leader's last log index. Upon achieving this match, the replicas update and commit their databases based on the synchronized log entries. This ensures consistency and integrity across the replica nodes.