Atypon/Wiley Inc, Amman, Jordan
Java and DevOps Bootcamp (Fall 2022)
Capstone Project
Decentralized Cluster-Based NoSQL DB System

A NoSQL database (DB) is a database that stores data differently from traditional relational DB systems. Thus, NoSQL DBs provide their own APIs for performing queries, instead of using SQL. A cluster-based NoSQL DB is a collection of nodes that can service multiple users, where each node has a replica of the DB. NoSQL DB clusters can have a manager node that works as a central point in the system. We refer to such systems as centralized systems. On the other hand, NoSQL DB systems can be decentralized, where there is no manager node, and instead the nodes in the DB rely on sophisticated schemes for ensuring data consistency and load balance. In this project, you are required to use Java to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster. Below, we first describe the application requirements, then we discuss important highlights in the implementation.

Application requirements

- Initially, there must be a bootstrapping step, which is responsible for starting up the cluster and initiating all nodes. More specifically, there is a “bootstrapping node” that provides initial configuration information (e.g., user information, Node IP addresses, etc) to all nodes so that they may communicate and function correctly. Note that the bootstrap node will also map users to nodes in a load-balanced manner. Users will use their assigned nodes for login.
- If there is a new user, then this user will first communicate with the bootstrapping node to obtain login information and its assigned node. There should be no communication between the user and the bootstrapping node afterwards.
- The bootstrapping node has no role beyond starting up the cluster, and functioning as an entry point for new users.
- Nodes should be capable of verifying login information for users to minimize security risks.
- After successfully logging to their assigned nodes, users can now send queries to the database.
- The database is a document-based database that uses JSON objects to store documents.
- Each DB has a schema, and each document within a DB has a JSON schema that belongs to the DB schema.
- DB queries should include creating or deleting a DB, creating or deleting a document within a DB, and reading or writing specific json properties within a document.

- A document (i.e., json object) has an ID which should be unique and indexed in an efficient manner.
- Create indexes on a single json property.
- Data, schemas, indexes are replicated across all nodes, i.e., stored inside the local file system of each node.
- Read queries are satisfied by any node. However, write queries have “node affinity”, i.e., a node where the write to a particular document can occur.
- If there is a write query to a document inside a node in which it has no affinity, then this node will send this write query to the node which has affinity.
- After a write is performed, the affinity node must “broadcast” this change to other nodes. If there was a read on any node during this time, then this read will read the old copy of the data. However, when a node is updated, then reads must return the updated copy of the data.
- Document-to-node affinity must be load-balanced.
- You may assume that reads to JSON properties are the majority of transactions, while writes are rare.
- There is at least one pre-determined DB admin

Implementation Highlights

- In your code, each node must be represented as a virtual machine (VM). Therefore, you will need mechanisms for sending/receiving data between VMs. For efficiency, use docker containers, which will communicate via a docker network.
- Under no circumstance are you allowed to use shared file systems or introduce any “centralization” in your implementation.
- Within the same node, use threads to serve users independently.
- An interesting race condition scenario is when multiple writes to the same JSON property occur simultaneously. To resolve this issue, use the “optimistic locking” scheme, in which a write will attempt to update the JSON property only if the current version of the data matches its own version. If there is a mismatch, then the write will fail and restart again after updating its version of the data.
- For JSON property indexing, you are not allowed to use existing indexing libraries such as Apache solr. Instead, we want you, the student, to implement the indexing.
- You may assume that the cluster has a fixed number of nodes, i.e., N, which can be configured during testing. In reality, clusters have autoscaling mechanisms, which will be ignored in this project for simplicity.

- Sending broadcast messages can cause congestion in a network. However, for simplicity, we will assume a small-size DB cluster. Thus, feel free to implement simple network broadcast mechanisms.
- Cluster Nodes often use caching techniques to make read queries faster. However, we will consider cache implementation to be an optional feature in the project.
- Create a demo application that is either a command line application or a web application that utilizes the database (e.g. email application, banking application, etc). Make sure to present evidences of correctness and load balance in your demo. Note that we really do care about how you tested your code.

What you need to submit:

- Submit your code
- Submit a report that includes information about:
  - DB implementation
  - The data structures used,
  - Multithreading and locks,
  - Data Consistency issues in the DB
  - Node hashing and load balancing
  - Communication protocols between nodes.
  - Security issues
  - Code testing
  - Defending your code against the Clean Code principles (Uncle Bob).
  - Defending your code against “Effective Java” Items (Jushua Bloch)
  - Defending your code against the SOLID principles
  - Design Patterns
  - DevOps practices