Name : Rishabh Agarwal
X500 : agarw266

Key-Value Store Implementation

Design Decisions and Justification:

1. Data Storage: The key-value store uses an in-memory data store represented by the `data` dictionary. This allows for fast read and write operations.

2. Data Persistence: Data persistence is implemented by periodically saving the `data` dictionary to a JSON file named "data.json." This design choice ensures that data is not lost in case of server shutdown or crashes. The periodic saving mechanism is implemented in a separate thread to avoid blocking the main server thread.

3. Multithreading: Multithreading is used to handle multiple concurrent client requests. Each client connection is managed by a separate thread (`client_handler`). This design choice ensures that the server can handle concurrent operations without blocking.

4. Locking Mechanism: To prevent race conditions and ensure data consistency, a thread lock (`data_lock`) is used when performing PUT and DELETE operations on the `data` dictionary.

5. Network Communication: The server listens on a specified port (12345) for incoming client connections. The communication between the client and server is done using simple text-based commands.

Challenges Faced During Implementation and Solutions:

1. Concurrency: Managing concurrent access to the `data` dictionary was a challenge. To overcome this, a lock (`data_lock`) was used to ensure that only one thread can modify the dictionary at a time, preventing data corruption.

2. Data Persistence: Implementing a data-saving mechanism that periodically saves data to a file without blocking the main thread required the use of multithreading. A separate thread (`data_saver_thread`) was used to periodically save data to a JSON file.

Assumptions Made:

1. The server uses a simple text-based protocol where the client sends commands like "GET key," "PUT key value," and "DEL key." It assumes that clients will follow this protocol.

2. Error handling is minimal in the code. The code assumes that clients will send valid commands and that the server's main purpose is to store and retrieve key-value pairs.

Potential Improvements and Features for Future Versions:

1. Error Handling: Implement robust error handling to gracefully handle unexpected client behavior or invalid commands.

2. Security: Enhance security by implementing authentication and authorization mechanisms to control access to the key-value store.

3. Data Compression: Implement data compression techniques to reduce the storage size of the JSON file, especially if the data store becomes large.

4. Scalability: Consider implementing a distributed version of the key-value store to handle larger datasets and improve scalability.

5. Query Language: Develop a more advanced query language for clients to perform complex operations on the data store.

6. Logging: Add detailed logging to record client operations and server activities for debugging and auditing purposes.

7. Configuration: Allow for configuration of the server's port and data file path to make it more adaptable to different environments.

8. Data Validation: Implement data validation to ensure that keys and values meet specific criteria or constraints.

9. Data Expiration: Add support for key-value pairs with expiration times, automatically removing data that is no longer needed.

10. Metrics and Monitoring: Implement monitoring and metrics collection to track server performance and usage statistics.

11. Replication and Failover: Consider adding replication and failover mechanisms to ensure high availability and data redundancy.

12. REST API: Offer a RESTful API in addition to the text-based protocol for more versatile client interactions.

In summary, while the provided code is a basic implementation of a key-value store with data persistence and concurrency handling, there are numerous opportunities for improvement and the addition of advanced features to make it more robust and versatile for real-world usage.