Reflection Report

Explain how you implemented concurrency to handle multiple

clients.

- To handle multiple clients concurrently, I used **goroutine**. Each client connection was processed in its own goroutine, allowing the server to handle multiple client requests simultaneously without blocking.
- Upon accepting a new client connection using **net.listener** a goroutine was spawned to manage the client session.
- Each goroutine was responsible for reading commands from the client, processing them, and sending responses back.

Describe the challenges you faced and how you solved them.

Synchronization Issues

- **Challenge**: Since multiple clients could simultaneously access and modify the shared key-value store (which is on the postgres database), race conditions could occur, potentially leading to inconsistent data.
- **Solution**:
    - I used a **sync.Mutex** to lock critical sections of the code that accessed or modified the store. This ensured that only one goroutine could interact with the database at any given time, preventing concurrent write conflicts.

Handling Responses for LIST request

- **Challenge**: For LIST request the client was unable to read all the data in one go, leading to partial responses, in fact the client was able to see the fist key value pair only.
- **Solution**:
    - On the server, I ensured that **LIST** sent responses separated with coma and at the end I added a new line so that the client can use it as a delimiter to read.
    - On the client, I read the response using the new line as delimeiter, now it can read all the responses as one line(may be it is lack of knowledge about the language and may have a better solution).

Discuss how you ensured data consistency when multiple clients

accessed the store concurrently.

To maintain data consistency when multiple clients accessed the key-value store concurrently:

- **Database Locking**: I used **sync.mutex** to guard all database operations, ensuring that only one client could execute a query or modification at a time.
- **Atomicity of Operations**: Each command (**PUT, GET, DELETE, LIST**) was treated as an atomic operation. By locking the database before executing any query, I prevented conflicts such as one client reading stale data while another was updating it.
- **Transaction Isolation**: While the Mutex prevented race conditions, the underlying database (Postgresql) ensured ACID compliance for individual operations, providing an additional layer of safety.