Write a basic key-value DB implementation. Program will accept DB commands as inputs from the command line and process them by creating DB structure in memory. The output will be displayed on the standard output stream.

Expectations:

- Choose any programming language.
- Test drive the code (use TDD). If not possible, write tests after the code is written.
- Write Readme describing the Assumptions / Technical decisions / future todos / known issues of your implementation.
- Aim of this exercise is to write modular and extensible code. It's okay if it is NOT highly performant, as we don't plan to use it in production.
- As with all things in software, the requirements will change over time. Good modular code is open for extension and closed for modifications (Open-Closed pricinple in SOLID). Aim to write such code.
- Don't get fancy, keep things simple and stupid. When in doubt, make reasonable assumptions and document them in the Readme.

Sample input is of format: `COMMAND ARGS...`

Case 1: SET key-values and then GET them. You can also DEL a key.

```
SET key1 value1
SET key2 value2
GET key1 // should return value1
DEL key2 // deletes key2 and its value
GET key2 // should return nil
SET key2 newvalue2
```

Case 2: Basic numeric operations and error cases

```
SET counter 0
INCR counter // increments a "counter" key, if present and returns incremented
GET counter // returns 1
INCRBY counter 10 // increment by 10, returns 11
INCR foo // automatically creates a new key called "foo" and increments it by
```

Case 3: Command spanning multiple sub-commands

Example 1: Happy path

```
MULTI // starts a multi line commands
INCR foo // queues this command, doesn't execute it immediately
SET key1 value1 // queues this command, doesn't execute it immediately
EXEC // execute all queued commands and returns output of all commands in an a
```

Example 2: Discard

```
MULTI // starts a multi line commands
INCR foo // queues this command, doesn't execute it immediately
SET key1 value1 // queues this command, doesn't execute it immediately
DISCARD // discard all queued commands
GET key1 // returns nil as key1 doesn't exists
```

Case 4: Generate compacted command output

Example 1:

```
SET counter 10
INCR counter
INCR counter
SET foo bar
GET counter // returns 12
INCR counter
COMPACT // this should return following output

SET counter 13
SET foo bar
```

Example 2:

```
INCR counter // returns 1
INCRBY counter 10 // returns 11
GET counter // returns 11
DEL counter // deletes counter
COMPACT // this should compact to empty output as there's no keys present in tl
```

Story 5: Expose the program via TCP and allow separate client programs to connect to it

So far, the program works by running on a terminal. Now, update the program to expose it via a TCP server. The server component should listen on a TCP port (example 9736). One or more clients can connect to the server via the port, issue commands and get results of the commands.

Write a separate client program that will run as a separate process and connect to server on the port (9736). Upon startup, the client program will try to connect to server running on TCP port 9736. The client will issue various command you have already implemented. For the client, implement, one more command - `DISCONNECT` to disconnect from the server (cleanly close the TCP connection) and exit.

Can you identify any potential problems when multiple clients issue concurrent requests to the server? Describe in the readme, how you will solve this problem.