ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ + ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

M111
Big Data Management

Project
Programming Assignment

Kylafi Christina-Theano

LT1200012

April, 2021

TABLE OF CONTENTS

Introduction

Overview

In this project, a simple version of a distributed, fault-tolerant KV DataBase is simulated.
All functions described in the instructions were implemented, numerous checks are performed to ensure the smoother execution of the process, multiple assumptions were made that are mentioned below (**Assumptions**) and there are descriptive comments throughout the code, explaining the way each function works and its role in the programme.

File Structure

File Kylafi_ChristinaTheano_Project.zip contains:

- "**Data**" folder:
    1. "**keyFile.txt**": space-separated list of key names and the respective data types
    2. "**nameFile.txt**": name dataset to get real "name" key values
    3. "**serverFile.txt**": list of server ip address-port pairs

- "**Scripts**" folder:
    1. "**trie.py**": the trie structure (Class) and the functions used to manage the key storing, retrieving, deleting, etc.
    2. "**createData.py**": creates the file of the entries to be stored by the servers later ("dataToIndex.txt")
    3. "**createData_funs.py**": the functions used in "createData.py"
    4. "**kvBroker.py**": starts all the servers from the "serverFile.txt" (a thread for each), stores each entry of the "dataToIndex.txt" file to k-random servers and then runs queries (retrieval / deletion) on all servers till the user quits the programme
    5. "**kvBroker_funs.py**": the functions used in "kvBroker.py"
    6. "**server_generator.py**": the server class ("Server_Generator") and its methods used in order to start/maintain/close the connection and terminate the respective server thread
    7. "**kvServer.py**": starts a server that connects to a given ip-address and port and during the execution of the programme, it stores/retrieves/deletes entries from its database till the user quits, when it deletes the trie structure created to store the entries and exits
    8. "**kvServer_funs.py**": the functions used in "kvServer.py"
    9. "**customExceptions.py**": custom exceptions used in the programme

- "**Docs**" folder:
    1. "**README.txt**": short instructions/details on how to run the code
    2. current report

Functionality

After running the code, the following **commands** are available for use:

1. **GET** < top_level_key > : requests the “top_level_key” key given by the user from all the servers that are still up. If k servers are down, a warning message is printed.

2. **QUERY** < top_level_key(.nested_key.nested_key… nested key) > : requests the keypath (top_level_key(.nested_key.nested_key… nested key)) given by the user from all the servers that are still up. If k servers are down, a warning message is printed.

3. **DELETE** < top_level_key > : deletes the “top_level_key” key from every server, if none of them are down, otherwise the command cannot be performed

4. **exit** (case insensitive) : quits the programme

Assumptions

1. In case of **duplicate keys at the same nesting level** (although the way the dataset is created does not allow such cases), only the latter is stored.

2. In case of **duplicate top level keys** (although the way the dataset is created does not allow such cases) between the different servers (in the same server only the first one is stored), the Broker checks how many different entries they have been retrieves and prints them all for the user.

3. Top level keys are case sensitive (e.g. both “person1” and “Person1” will be stored as different entries - Oracle way contrary to the SQL way).

4. The separator between the command and the key/keypath can be either 1 or multiple spaces/tabs (checks ensure the right command parsing in either way).

5. After the query command (GET/DELETE/QUERY), only 1 key or dot separated keypath string is allowed (checks are performed).

Experimenting

1. In order to crash test my code, I have tried to create big data files (-d 999, -m 27, -l 1000, -n 100, etc.), in which the programme performed exceptionally

2. I have also tried to include cases where the connection is not permitted (commented out “ **self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)** ” in “server_generator.py”) or there is a problem with the socket in general, to catch and manage such kinds of emergencies (proper error messages and actions - might not have thought of each an everyone of them though..)

Data Creation

Implementation Details

- The process of data creation starts with the following (also explained in the “README.txt file) :
  - createData.py [-h] [-k K] [-n N] [-d D] [-m M] [-l L]
- Each top level key’s value dictionary can be either an empty set or a set of key-value pairs
- Each top level key has a probability of 10% to have an empty set as value
- Each nested key’s value can be either an empty set, a set of key-value pairs, an integer, a float or a string
- Each nesting key has a probability of 20% to have an empty set as value
- The random string generator generates strings that contain uppercase, lowercase ascii letters and digits
- The range of the values are the following:
  - int : 1 - 200
  - float : 0.00 - N, where N is a random number in range 1 - 100 (rounded to 2 decimal places)

Entry File Format

Each entry has the following format in the “dataToIndex.txt” file :

```
“top_level_key1” : { “key1_1” : { “key1_11" : “value1_11” ; “key1_12” : “value1_12” } ; “key1_2" : “value1_2” }
“top_level_key2” : { “key2_1” : { “key2_11" : “value2_11” } ; “key2_2” : “value2_2” } ; “key2_3" : “value2_3” ; “key2_4" : “value2_4” }
.
.
.
```

which means:

- the keys are in double quotes with no space between the quotes and the key
- there is a single space between brackets-keys, brackets-semicolons, keys-colons, etc
- a newline “\n” character is used as a separator between the entries

Key Value Store

Connection Information

In order to perform smooth and successful data exchange between the Broker and the servers, sockets based on TCP protocol are used. Also, there are two custom messages indicating the end of each exchange or the server exit ("#__DONE__#" and "#__exit__#" respectively). Function **socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)** is also used to avoid connection refusal due to "socket already in use" error, in cases of immediate rerun of the programme.

KV Broker

When the user starts the Broker (kvBroker.py [-h] [-s S] [-i I] [-k K]) then the servers also start (a thread for each), the socket connections are established (unless problems occur) and then the Broker starts sending each entry from the "dataToIndex.txt" file to k randomly picked servers each time to store it. Then, after the process of entry storing is finished, the Broker waits for user input (queries) to make to the servers. If k servers or more are down, a warning message is printed to the user, indicating the lack of result reliability because of the servers that are down, however it continues to work till the user types "exit" (case insensitive). In case of wrong input format or other errors, the Broker outputs proper messages. The Broker performs regular checks on servers as well as detailed examination of the queries aiming for a less complicated server workload.

KV Server

When a server starts from the Broker (server-thread), then it is associated with a certain ip address and port that was given as an argument from the Broker. A socket connection is established and then communication between the server and the Broker starts. In the beginning, the Broker sends all the servers a message indicating the buffer size that will be used for data exchange. Then, the entry storing phase begins during which each server stores an entry at a time in each trie structure (server's database described below). Last but not least, after the previous procedure, the servers are listening to the aforementioned sockets (ip-port pairs) for queries, and perform the proper actions for entry deletion/retrieval or programme/thread exit.

Trie Structure

Each server has its own database/entry storage structure. It is implemented as a trie. I have designed this structure as a linked tree of Trie_Node() class instances, where the top level keys are stored under the root trie node and then at each top level key's last node, a structure of nested trie structures implements this key's payload (value dictionary/set). In more detail concerning the nested trie structure, at each of the top level key's nesting levels, every key with another nesting level as value, has its own nested trie structure, implementing exactly that nesting. The keys with a simple value such as int, float or empty set, don't have a nested trie structure "below" them. For the previously mentioned structures to be created/managed/deleted, multiple variables and methods/functions were implemented, with comments in the code for details on the procedures.

References

1. **data/nameFile.txt**
https://www.usna.edu/Users/cs/roche/courses/s15si335/proj1/files.php%3Ff=names.txt.html