Programming Assignment

In this project we will be creating a simple version of a distributed, fault-tolerant, Key-Value (KV) database (or store), with a few tweaks. From Wikipedia: a key-value database, or key-value store, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, and a data structure more commonly known today as a dictionary or hash table. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to find the data within the database. In our case, we will be using a Trie instead of a hash table for storing the keys.

You can use any language of your choice but you need to implement the Trie data structure used in the KV-Server below from scratch.

1. Data Creation (20%)

Our KV store will be able to index data of arbitrary length and arbitrary nesting of the form:

```
key : value
```

In this case, `key` represents the key of the data that we care to store and value is the payload (or value) of the data that we want to store for that key. The `value` can also contain a set of `key : value` pairs. Here is an example of some data that we are interested in storing in our KV store:

```
"person1" : { "name" : "John" ; "age" : 22 }
"person2" : { "name" : "Mary" ; "address" : { "street" : "Panepistimiou" ; "number" : 12 } }
"person3" : { "height" : 1.75 ; "profession" : "student" }
"person4" : {}
```

As you can see, the payload (or value) is also a set of key-value pairs and can be nested. For example, person2 above, has another set of key values as the values of its subkey "address". person4 has an empty value. We only allow records of the form above, i.e. we either have an empty value or a value with key-value pairs. For example the following data is incorrect:

```
"person5" : "hello"    <-- wrong value is not key:value or {}
"person6" : { "address" : { "there" } } <-- wrong, value inside address is not key:value or {}
```

For each value, we can store either an empty Integer (e.g. 12), a Float (e.g. 12.5), a String (e.g. "hello", a set of Key Value pairs (e.g. { "key1" : 5 ; "key2" : "five" }, or an empty set of KV value pairs (i.e., {}). Each key is of type String only.

Your first task will be to write a program that generates syntactically correct data that will be loaded to your key value store. Your program should operate as follows:

```
createData -k keyFile.txt -n 1000 -d 3 -l 4 -m 5
```

where
`-n` indicates the number of lines (i.e. separate data) that we would like to generate (e.g. 1000)
`-d` is the maximum level of nesting (i.e. how many times in a line a value can have a set of `key : values`). Zero means no nesting, i.e. there is only one set of key-values per line (in the value of the high level key)

`-m` is the maximum number of keys inside each value.
`-l` is the maximum length of a string value whenever you need to generate a string. For example 4 means that we can generate Strings of up to length 4 (e.g. “ab”, “abcd”, “a”). We should not generate empty strings (i.e. “” is not correct). Strings can be only letters (upper and lowercase) and numbers. No symbols.
`-k keyFile.txt` is a file containing a space-separated list of key names and their data types that we can potentially use for creating data. For example:

```
name string
age int
height float
street string
level int
...
```

The goal of the program is to randomly generate as much data as was specified by the `-n` parameter, up to the given length (`-l`) of strings, up to `(-m)` keys per nesting level and up to the given nesting level (`-d`). Do not worry if the data do not make sense (e.g. age contains an address). This task is meant for you to create datasets that you can use to develop the remaining of the project. You can use random key names and types for the keyFile.txt if you like and you can create your own keyFile.txt. For **only** the top-level keys you can either generate random strings or you can generate keys of the form key1, key2, key3, etc. for easier debugging.

2. **Key Value Store (80%)**

This part consists of two programs, a Key-Value broker that will be accepting queries and will be redirecting requests to the Key-Value servers, collecting the results and presenting them to the user and a KV Server that will be storing the actual data and will be handling the queries coming from the broker. We describe each of these modules next.

2-a) KV Broker (20%)

The broker should start with the following command:

```
kvBroker -s serverFile.txt -i dataToIndex.txt -k 2
```

The `serverFile.txt` is a space separated list of server IPs and their respective ports that will be listening for queries and indexing commands. For example:

```
123.123.12.12 8000
123.123.12.12 8001
123.2.3.4 9000
```

Is an example of a `serverfile` indicating that this broker will be working with 3 servers with the IPs described and on the respective ports described.

The `dataToIndex.txt` is a file containing data that was output from the previous part of the project that was generating the data.

The `k` value is the replication factor, i.e. how many different servers will have the same replicated data.

Once the kvBroker starts, it connects to all the servers, and for each line `dataToIndex.txt` it randomly picks `k` servers where it sends a request of the form PUT data. For example, for the first line above in the first part of the project it will send the following command to each one of the k servers (there needs to be a white space after PUT, but all other whitespace is flexible):

```
PUT "person1" : { "name" : "John" ; "age" : 22 }
```

Each of the servers now stores (in-memory) the data that was sent over the socket. If everything was successful it should respond to the broker with `OK` or `ERROR` if there was a problem.

Once the indexing process has completed, the broker now expects from the keyboard one of the following commands:

a) `GET key`

In this case, the data with the given high-level key (i.e., you won't be searching inside any value) is queried across **all** servers and if the results are found it is printed on the screen. For example:

```
GET person1
```

Should query all three servers of the example above and print:

```
person1 : { name : John ; age : 22 }
```

```
GET name
```

Should return `NOT FOUND` as `name` is not a high-level key.

You are free to handle quotes "" as you like. Specifically, the following input and output is also correct, but please specify how you handle this (as well as any other assumptions you make) in your README:

```
GET "person1"
"person1" : { "name" : "John" ; "age" : 22 }
```

Since we implemented k-replication, the broker should continue to work unless k servers are down. For example, for k=2 if we had 3 servers running and one is down (i.e., 2 left) the server can still compute correct results. If >= 2 servers are down the broker should output a warning indicating that k or more servers are down and therefore it cannot guarantee the correct output.

b) `DELETE key`

This command deletes the specified high-level key (i.e., you don't need to search within each value). This command needs to be forwarded to all servers. If there is even one server down, delete cannot be reliably executed and thus there should be a message indicating that delete cannot happen.

c) `QUERY keypath`

This command is similar to GET above but is meant to return the value of a subkey in the value part of the high level path. For example, for the data

```
"person2" : { "name" : "Mary" ; "address" : { "street" : "Panepistimiou" ; "number" : 12 } }

QUERY person2
```
should return (i.e. the same as GET)
```
person2 : { name : Mary ; address : { street : Panepistimiou ; number : 12 } }

QUERY person2.name
```
should return
```
person2.name : Mary

QUERY person2.address.number
```
should return
```
person2.address.number : 12
```

Both `GET` and `QUERY` should specify that the key was not found if a query with a non-existent key was asked. `QUERY` works in an identical way to `GET` as far as replication (i.e., number of available servers) is concerned.

2-b) KV Server (50%)

The KV Server will be starting at a specific IP and port and will be serving queries coming from the KV broker. The KV Server should start as follows:

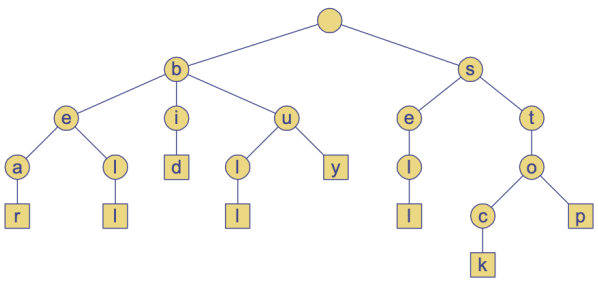
```
kvServer -a ip_address -p port
```

The server starts at the specified IP address and port (which should be one from the server file that the broker is accepting as input) and is waiting for queries. Once the query is received (as described in the broker section above), the server parses the query. If the query is incorrect (e.g. missing }) the server returns ERROR to the broker together with a message describing the error. If the query is correct, the server looks up its internal data structures and attempts to find the data corresponding to the query. If the data is found, it is returned. If the data is not found, then NOTFOUND is returned.

In order to search the data internally (in-memory) efficiently the server maintains a data structure called a trie (see here for more details) where the top-level keys are stored. When a query comes in, the server uses the trie to identify whether the key exists, and if so, follows the path where the values are stored. In the case of a `QUERY` command, the values at the final level are searched linearly (take care of the nesting though) or using any other data structure that you like.

Here is one example of a trie for a set of keywords:

Example: standard trie for the set of strings
S = { bear, bell, bid, bull, buy, sell, stock, stop }



For the key “bear” we would be looking for the data in the leftmost branch of the tree.

2-c) Using Only a Trie (10%)
The final 10% for the project is for using only the trie data structure for indexing and searching all of the keys (both top-level and within the values and the nesting) and no other data structure.

Deliverables

- All files for your project need to be zipped in **one single zip file.** The file name should be in the following format: Lastname_Firstname_Project.zip
- Please email the zip file to the instructor by the deadline. **There are absolutely no exceptions and no extensions.**
- Inside the zip file, please include a) your source code b) any results that are part of the project c) a README text file that explains how to compile (if needed) and run your code and d) a short and concise report (preferably pdf - no more than 10 pages) describing your work and your approach to solving the project. **Please do not include binaries, nor the original or any derived data in the zip file.**

Important Notes

- Please check the class Website: http://www.di.uoa.gr/~antoulas/m111 regularly for announcements and/or clarifications to the project. Announcements will show up there and will also be sent to the mailing list.
- You are free to make any assumptions you may need to along the way as long as you document them in your report.
- The project is meant to be worked on by the student submitting and only that student. In the event that there is a submission not worked on by the student, then the student fails the class.
- Copying code from existing sources (e.g., the Web, github, etc.) is strictly not allowed.
- You are responsible to safeguard your code, so if you are using a code repository service (e.g., github) keep your repository private. Cases when code was cloned and was also submitted by someone else cannot be reliably traced and result in everyone involved (regardless of who is at fault or who copied from whom) failing the class.
- Although students are expected (and encouraged) to chat among them on potential solutions and approaches, sharing code and solutions is strictly not allowed. In the event that two or more students provide submissions that have common pieces, everyone involved (regardless of who is at fault or who copied from whom) fails the class.