Key-Value Store with History

Introduction

The goal of this coding exercise is to design and implement a key-value store with history. Like a traditional key-value store, this store should support the following APIs:

1. **`get(key)`**: returns all values associated with the key, if present.
2. **`put(key, value)`**: adds or updates the key with the value.
3. **`del(key)`**: deletes the key from the store.
4. **`del(key, value)`**: deletes the specified value from the key.

In addition, this store should also support the following APIs:

3. **`get(key, time)`**: returns all values associated with the key up to the specified time.
4. **`diff(key, time1, time2)`**: returns the difference in value associated with the key between time1 and time2. **`time1 <= time2`**.

Here is an illustration of the key-value store:

```
Time: 0           1           2           3           4           5
      ---------------------------------------------------------------------
      |           |           |           |           |           |
APIs: put("A","c") put("B","d") get("A")   put("A","e") get("A")   get("A", 2)

result:                       ["c"]                   ["c","e"]   ["c"]


Time: 6           7           8           9           10          11
      ---------------------------------------------------------------------
      |           |           |           |           |           |
APIs: del("A")    get("A")    get("A",5)  put("B","f") del("B","d") get("B")

result:           []          ["c","e"]                           ["f"]
```

Example of diff API:

```
diff("A", 1, 2) returns []
diff("A", 3, 5) returns []
diff("A", 1, 4) returns ["e"]
diff("B", 0, 1) returns ["d"]
```

System Design

Design and implement the key-value store as a Service. The key-value store implements the friend-list in a social network, where key is the username and value is a list of names of the friends of the user (key). The value could change over time for given user and this key-value store captures this dynamic nature of the social network. Note that the **`get(key, time)`** and **`diff(key, time1, time2)`** should retain the order in which the values are added to the given key.

As an optional bonus assignment, design and implement a caching layer for the key-value store. The caching layer provides the same set of APIs as the key-value store.

The service layer and the caching layer could be implemented as a web-service, thrift service (https://thrift.apache.org/), grpc or a pure socket service. Implementing a thrift service is a bonus.

Also, provide client code that issues requests (puts and gets) to the service (either directly if no caching layer is implemented or using the caching layer).

Implementation Notes

- No further clarifications will be entertained about the problem statement.
  - Make reasonable assumptions and explicitly state them in your solution (in the form of code comments)
- Provide detailed code-comments. No additional documentation is necessary.
- Languages accepted: Java, Scala, C, C++, Go, Python
- We look for the following in your submission:
  - Space and time complexity of get and put operations
  - Error handling
  - Scale
  - Unit tests
  - Code readability
- Submit only the code as an attachment to an email or a github link.