Supercell 2023 SWE Intern Exercise

1 Background

Assume we want to build a social network on top of many mobile games. Users of such a network would like to see what their friends are doing. For example, we might want to be specific like ”is fighting a boss” or we might want to track their weaponry and avatars. To track such information games will write updates to a custom key-value store. An update would be a JSON of the following form:

```
{
  "type": "update",
  "user": "12-3456-789",
  "timestamp": 1667462835,
  "values": {
    "weapon": "sword",
    "armor": "heavy"
  }
}
```

The key-value store will then broadcast any updated values to all the friends of the user ”12-3456-789”.

As with any distributed system, updates can be lost, duplicated or reordered. It might therefore be possible that a previously received update had a later timestamp. In that case, we should not broadcast the received value to the user’s friends as the value is not current.

2 First Problem

Your job is to implement the state tracking and broadcasting system outlined above. The first problem is to write an application that accepts an input file (see section 2.1 for format) and outputs to stdout.

The input supports three possible commands:

1. Mark two users as friends.
2. Unmark two users as friends.
3. Update some state entries of a user.

A command that marks two users as friends has the following format:

```
{ "type": "make_friends, "user1": "12-345", "user2": "67-890" }
```

This will make ”67-890” friends with ”12-345”. The relation is symmetric. Similarly, to unmark these two users as friends, the command would be:

```
{ "type": "del_friends, "user1": "12-345", "user2": "67-890" }
```

We don't add timestamps to the commands that mark and unmark friends. You can assume they will come in the right order.

A state update is a JSON of the following form:

```
{
  "type": "update",
  "user": "12-3456-789",
  "timestamp": 1667462835,
  "values": {
    "weapon": "sword",
    "armor": "heavy"
  }
}
```

There can be any number of values in a state update. Note that the ”values” map has string-typed keys and values.

2.1 Input Format

The input consists of multiple JSONs one per line as follows:

```
{ "type": "make_friends", "user1": "12-345", "user2": "67-890" }
{ "type": "make_friends", "user1": "ab", "user2": "12-345" }
{ "type": "update", "user": "ab", "timestamp": 100, "values": { "foo": "bar" }}
{ "type": "update", "user": "ab", "timestamp": 99, "values": { "foo": "bar", "baz": "crux" }}
{ "type": "update", "user": "ab", "timestamp": 101, "values": { "foo": "bar" }}
{ "type": "del_friends", "user1": "12-345", "user2": "67-890" }
```

2.2 Output Format

The output format consists of multiple JSONs one per line (listing the set of broadcasts made:

```
{ "broadcast": [ "12-345" ], "user": "ab", "timestamp": 100, "values": { "foo": "bar" }}
{ "broadcast": [ "12-345" ], "user": "ab", "timestamp": 99, "values": { "baz": "crux" }}
{ "broadcast": [ "12-345" ], "user": "ab", "timestamp": 101, "values": { "foo": "bar" }}
```

Note that we only broadcast the values in the update that had a newer timestamp than the values currently stored. If there are no values to broadcast or the user has no friends, then no broadcast message is produced. A single update should produce at most one broadcast JSON, so all values and friends are collected into a single broadcast message.

2.3 Test Cases

You can find test cases to download from `https://supr.cl/swe-2023-tests`

3 Second Problem

A status service like the one outlined above would in the real world get deployed as an HTTP server. Such a server would receive concurrent updates from multiple game servers. Assuming 10 million concurrent players and each of them generating one state update every 10 seconds, the status service would receive one million updates per second. An actual implementation would therefore need to be sharded to multiple servers. However, you do not need to worry about a distributed implementation or writing an HTTP server.

The second problem asks you to simulate a real-world situation where requests are handled concurrently. Your job is therefore to make your solution to the first problem thread-safe. You should write an application that reads an input file as in section 3.1 and applies the updates in parallel using multiple threads. Output should go to stdout.

Each entry in the output should have the value of its update with the latest timestamp. What we are looking for are readable and efficient implementations that support high levels of concurrency.

3.1 Input Format

The input format is the same as in the first problem except that there are only ”update” commands.

3.2 Output Format

Your output should be a single JSON file containing the final state of all players. As an example, if you have two players ”ab” and ”12-345”, then their state would be output as follows:

```
{
  "ab": {
    "foo": "bar",
    "baz": "crux"
  },
  "12-345": {
    "aaa": "bbb"
  }
}
```

Here ”ab” had two state entries ”foo” and ”baz” with the values above while ”12-345” had an entry ”aaa”.

You do not need to output the JSON without line breaks as in the first problem, i.e. you’re free to pretty print it for readability.

3.3 Test Cases

You can find test cases to download from `https://supr.cl/swe-2023-tests`

3.4 Extra Credit

For extra credit, you can also submit a benchmarking program that concurrently applies updates at a high rate. It’s up to you how you want to benchmark your code and to think about what’s relevant to benchmark. If you have access to machines with many cores, how does it scale?

4 What We Are Looking For

We want to see correct, clear, simple and efficient solutions. You should include some tests that show you have thought about what the corner cases are.

5 Submitting

You should package your solution as a zip file. Your solution needs to include instructions for building your programs on either Linux or MacOS. Both programs should be executable by writing:

```
my_program -i <input_file>
```

The solution to the second problem should also include a short description of why you implemented the solution the way you did, i.e. any assumptions you made.

You can write your solution in any language, but reviewers will probably be happier if you do it in one of C++, Java, C#, Rust, Go or Assembly (6510 preferred). For C++, the following single header JSON library might be useful:

https://github.com/nlohmann/json

Please see the Job Description on supercell.com/careers for details on the role and the application form.