The First Mandatory Programming Assignment:

The CORBA Taste Profile Service

INF5040 Autumn 2017

Objective

- To develop an object-based distributed system;
- Using the CORBA programming model and middleware;
- Following the simple client/server architecture;
- Handling large data sets in a distributed environment;

Scope

The system consists of a distributed musical taste profile service. The Application functionality is provided by a remote object residing at the server site. Client objects interact with the server through remote method invocations. The client can invoke the methods defined in the server's interface specification (IDL file). The IDL file is available at the course website and the students are required to download it, modify it according to the assignment specification and compile it using the **idlj** tool in JAVA SDK - to generate the code for stub, skeleton, and other artifacts of the CORBA architecture. Using the generated code, the students will develop the client and server code for the application.

The server will have access to a Musical Taste Profile data set. The data set can be downloaded from http://labrosa.ee.columbia.edu/millionsong/tasteprofile. The file (about 3GB, 500MB compressed) contains the taste profile of users collected from different music applications. The user profile information consists of which songs were played by the user and how many times each song was played (see echonest.com). The file format is as follows:

```
User ID:                                   Song ID:            Play count:
b80344d063b5ccb3212f76538f3d9e43d87dca9e   SOFZFQU12A8C13CAB8  1
b80344d063b5ccb3212f76538f3d9e43d87dca9e   SOGJAOS12A6D4F7459  1
b80344d063b5ccb3212f76538f3d9e43d87dca9e   SOHQIAG12A8C136F64  1
8937134734f869debcab8f23d77465b4caaa85df   SOAFPAX12AB0187A17  6
8937134734f869debcab8f23d77465b4caaa85df   SOEBOAR12A6D4FD136  1
8937134734f869debcab8f23d77465b4caaa85df   SOFRDND12A58A7D6C5  5
8937134734f869debcab8f23d77465b4caaa85df   SOJCGJJ12A8AE48B5D  5
e006b1a48f466bf59feefed32bec6494495a4436   SOGYNZW12A8C1388A4  5
e006b1a48f466bf59feefed32bec6494495a4436   SOIICEQ12A6D4F7FE0  1
e006b1a48f466bf59feefed32bec6494495a4436   SOIRRMU12A6D4FB0C0  2
```

The service you will implement is responsible for parsing this data set and provide clients with information about users' musical taste. The clients, using remote invocations, can ask the server how many times a given song was played by all users and how many times a given user played a given song.

Technical features

1. **Server**
   a. The service interface exposes the following methods that have to be implemented by a servant class (more details given as comments in the IDL file):
      i. **getTimesPlayed** – Given a song id, return the number of times that this song was played by all the users.
      ii. **getTimesPlayedByUser** – Given a song id and a user id, return the number of times the song was played by this user.
      iii. **getTopTenUsers** – Return a sorted list of ten users in ascending order by the total number of times the users have listened to songs.
   b. The servant class should be instantiated, registered with an unique name and started with a server application that implements the main() method.
      i. A HelloWorld example is available in the course website.
   c. In a distributed environment there is always communication latency. To simulate network latency, **pause the server execution for 60 milliseconds every time a remote method is invoked**. This way, you can test your code by deploying both client and server in the same machine.

2. **Client**
   a. The client code will parse an input file containing a sequence of queries. Each line specifies a method name and the arguments that should be invoked on the remote server. The input file will be provided at the course website. For each remote invocation, the client will print the result of the invocation and the time it took. The output should also be printed to a file.
   b. Input file format: `<method name>    <argument1>  <argument2>`
   c. Output file format example:

```
    Song SOUDSFN12A8C144B74 played 1069 times. (61 ms)
    Song SOJCPIH12A8C141954 played 11205 times. (61 ms
    Song SONKFWL12A6D4F93FE played 2 times by user
b64cdd1a0bd907e5e00b39e345194768e330d652. (62 ms)
```

Assumptions and Requirements

- It is a requirement that the server **cannot keep 3GB of data in memory**. As a first solution, you can create a naïve server implementation that parses the whole data set every time a request is made. This will significantly reduce the server performance in terms of invocation response time.
- We assume that clients are expected to follow a particular behavior: they will most probably query for songs and users that are popular. Based on your first solution, design a second, smarter implementation that keeps the most popular songs and users in server memory without violating the following memory requirements on the server:
  - There are around 400.000 song ids, and a Map<String, Integer> of (song ids, play count) does not require a lot of memory (about 10MB of memory). Therefore, it is acceptable to store this Map in server-side memory for method **getTimesPlayed.**

    - There are around 1.000.000 users and each user has played hundreds of songs. It is not possible to keep all this information in memory for method **getTimesPlayedByUser** without using more than 3GB of memory. For this assignment, **you can keep at most 1.000 user profiles in memory** (or about 20 to 30MB of memory).
    - Popular songs are the ones with highest play count across all users.
    - Popular users are the most active ones (user with highest total play count).
- The clients are also expected to perform queries about the same user with consecutive method invocations. Therefore, it might be a good idea to implement a different method in the server that outputs the complete profile of a user, so that the profile can be kept in a local cache at the client-side, preventing the overhead of consecutive remote invocations.
    - The signature of this method is similar to **getTimesPlayedByUser**, but should have an extra out argument that returns the complete user profile:
        - `User getUserProfile(in string user_id, in string song_id);`
    - The new method will require the creation of the **User** and **Song** valuetypes:

```
valuetype Song {
        public string id; public
        long play_count;
};


valuetype User {
        public string id;
        public sequence<Song> songs;
};
```

- The method **getTopTenUsers** will require the creation of valuetype **TopTen**, and an extra argument for the method to return the list:

```
valuetype TopTen
{
        public sequence<User> topTenUsers;
}


interface Profiler
{
        TopTen getTopTenUsers();

        . . . .
        . . . .
}
```

    - You should add the **getUserProfile** and **getTopTenUsers** methods and the **User**, **Song** and **TopTen** valuetypes to the CORBA idl file, recompile it and implement the method in the server code.

- Note about **Valuetype** structures in Java:
  - *The idlj compiler will generate abstract classes for the valuetype structures (such as User and Song). Concrete classes that extend the abstract ones (e.g. UserImpl and SongImpl) will have to be implemented in order to make the code work in Java.*

Measurements:

1. Using the file input.txt as the input for the client, run it and produce an output file containing the returned values from both the **getTimesPlayedByUser** and **getTimesPlayed** methods, without client-side user profile caching. Then use the **getUserProfile** method to store user profiles on the client, and generate a new output-file containing the results. What is the performance gain in terms of the total time that it takes to complete all the remote invocations? You will have to produce:
   - 1 output file without using **getUserProfile**;
   - 1 output file using **getUserProfile**;
2. Do not save the Map of songs and 1000 user profiles in the server memory. How much time does it take on average to provide a response to a remote method invocation? How does this naïve solution compare with the strategy using server-side memory? You will have to produce:
   - 1 output file with server memory disabled. (The client-side caching can stay on.)
   - If it takes too much time to parse the whole input file, stop after a few invocations. That should be enough to calculate the average invocation time.
3. An output file containing the results of the **getTopTenUsers** method.

Summary of measurements:

- Four output files.
- One without client-side caching of user-profiles, one with.
- One output file without server side memory disabled.
- One output file containing the results of the **getTopTenUsers** method.
- Format on the **getTimesPlayed** and **getTimesPlayedByUser** output files:

  -
    ```
    Song SOUDSFN12A8C144B74 played 1069 times. (61 ms)
    Song SOJCPIH12A8C141954 played 11205 times. (61 ms)
    Song SONKFWL12A6D4F93FE played 2 times by user
    b64cdd1a0bd907e5e00b39e345194768e330d652. (62 ms)
    ```
- Format **getTopTenUsers** output file:
  - `<userid>     <total of times listened to songs>`

Development Tools:

1. *Integrated Development Environment*: Eclipse IDE for **Java EE** Developers: https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar

2. ***CORBA****: Already integrated into Java Development Kit* a. **idlj** and **orbd** tools

We will accept minor deviations in the programming environment from the description above if agreed with the TA. Such deviations should be coordinated with the TA as soon as possible. Note that you also have to provide detailed instructions in your documentation on how to install and execute your software.

Deliverables:

Via the Devilry system:
A description of your design and implementation (Word, plain text, or preferably PDF), which should include a user guide for compiling and running your distributed application. The document should include screenshots of both client and server under execution.

Everything that is needed to compile and run your distributed application (in a zip archive), such as:

- Source code (should be well commented);
- Ready to deploy jar file;
- Output files containing the required measurements;
- Any other artifact that you consider relevant to compile and run your application.

On the day of submission:
Present your assignment to the TA by explaining your design and running the server and the client.

Deadline: 23:59 on October 9, 2017