The First Mandatory Programming Assignment:

The CORBA Taste Profile Service

IN5020 Autumn 2019

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 two Musical Taste Profile data sets. The data sets are available for download from the course website in a single zip archive named train_triplets (about 3GB, 1GB compressed). The files contain the song profiles collected from different music applications. The song profile information indicates which users played the song and how many times each user played it. The file format is as follows:

Song ID:	User ID:	Play count:
SOAAADD12AB018A9DD	8cf21d682f872dbe91296690359af2010e5195ca	3
SOAAADD12AB018A9DD	01926c22f16898fb986e2077625cefdc9f25bca3	1
SOAAADD12AB018A9DD	e9155469de206f847a66c7c5564f0bb5ae392708	1
SOAAADD12AB018A9DD	9949ae46ec7d99b2e2dc2a77150107107a937c18	1
SOMPBUE12AB017C4EF	7572a214256196c5b7e361a63d7b40c4344bdd35	1
SOMPBUE12AB017C4EF	4ba581f251b7e179e8f46cbfbf7c2f5d181d4f0f	8
SOMPBVF12A6D4F69CC	b9125e0934eba58b469db25dd9ec47065eca38d5	31
SOMPBVF12A6D4F69CC	c36e3af76247ff3c8173ff2f77910d4457bcfec3	7
SOZZZWN12AF72A1E29	3bee69341805736b847192f7725b3e003b2f3b1f	7
SOZZZWN12AF72A1E29	843af1c9f8e92133b1647794250f1b02428599cf	1

The service you will implement is responsible for parsing these two data sets and providing clients with analytical information about songs and users. 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. **getTopThreeUsersBySong** – Given a song id, return a list of top three users who have listened to the song the highest number of times. The list should be sorted in ascending order from the user ranked 3rd to the user ranked 1st.
iv. **getTopThreeSongsByUser** – Given a user id, return a list of top three songs played the highest number of times by this user. The list should be sorted in ascending order from the song ranked 3rd to the song ranked 1st.

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 80 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. (81
ms) Song SOJCPIH12A8C141954 played 11205 times.
(81 ms Song SONKFWL12A6D4F93FE played 2 times
by user
b64cdd1a0bd907e5e00b39e345194768e330d652. (82 ms)
```

Assumptions and Requirements

1. Naive implementation

- It is a requirement that the server **cannot keep 3GB of data in memory**. As a first solution, you can create a naive 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.
- The method **getTopThreeUsersBySong** will require the creation of valuetype **UserCounter** and valuetype **TopThreeUsers**, and an extra argument for the method to return the list:

```
valuetype UserCounter
{
        public string user_id;
        public long songid_play_time;
}

valuetype TopThreeUsers
{
        public sequence<UserCounter> topThreeUsers;
}

interface Profiler
{
        TopThreeUsers getTopThreeUsersBySong(string song_id);
        - - - -
        - - - -
}
```

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

```
valuetype SongCounter
{
        public string song_id;
        public long songid_play_time;
}

valuetype TopThreeSongs
{
        public sequence<SongCounter> topThreeSongs;
}

interface Profiler
{
        TopThreeSongs getTopThreeSongsByUser(string user_id);
        - - - -
        - - - -
}
```

2. Server-side caching

- 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, **SongProfile**> of (song ids, total_play_count, top_three_users) does not require a lot of memory (about 40MB of memory). Therefore, it is acceptable to store this Map in server-side memory for method **getTimesPlayed** and **getTopThreeUsersBySong.**
  - The **SongProfile** will be defined as follow:

```
valuetype SongProfile
{
        public long total_play_count;
        public TopThreeUsers top_three_users;
}
```

  - Note that: **TopThreeUsers** is defined as in the previous section **Naïve Implementation**.

- 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 methods **getTimesPlayedByUser** and **getTopThreeSongsByUser** 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).
- The **UserProfile** will be defined as follow:

```
valuetype UserProfile {
        public string user_id;
        public long total_play_count;
        public sequence<SongCounter> songs;
        public TopThreeSongs top_three_songs;
};
```

- Note that: **TopThreeSongs** is defined as in the previous section **Naïve Implementation**.
- 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).

3. Client-side caching

- 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 should be defined as follow:
  - ✐ **UserProfile** getUserProfile(in string user_id);

4. Changes to the idl

- You should add the **getUserProfile**, **getTopThreeUsersBySong, and getTopThreeSongsByUser** methods, and the **UserCounter**, **TopThreeUsers, TopThreeSongs, SongProfile, SongCounter** and **UserProfile** 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 UserProfile and SongCounter). Concrete classes that extend the abstract ones (e.g. UserProfileImpl and SongCounterImpl) 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 naive 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 stayon.)
    - 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 **getTopThreeUsersBySong** method.
4. An output file containing the results of the **getTopThreeSongsByUser** method.

Summary of measurements:

- Five output files: "clientside_caching_on.txt", "clientside_caching_off.txt", "naive.txt", "topuser.txt", "topsong.txt".
- "**naive.txt**" output file with server side memory disabled.
- "**clientside_caching_off.txt**" without client-side caching of user-profiles, "**clientside_caching_on.txt**" with client-side caching of user-profiles.
    - Note that: server caching is enabled for both cases.
- "**topuser.txt**" output file containing the results of the **getTopThreeUsersBySong** method.
- "**topsong.txt**" output file containing the results of the **getTopThreeSongsByUser** method.
- Format on the **getTimesPlayed** and **getTimesPlayedByUser** output files (Three first file):
    - **Song SOUDSFN12A8C144B74 played 1069 times. (81 ms) Song SONKFWL12A6D4F93FE played 2 times by user b64cdd1a0bd907e5e00b39e345194768e330d652. (82 ms)**
- Format on the **getTopThreeUsersBySong** and **getTopThreeSongsByUser** output files (The last two files):

  User <userid> played <total of times listened to songid> times.
  Song <songid> was played <total of times listened to songid> times.

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 September 27, 2019