Distributed Systems

IN5020

Programming Assignment 1

shielak, praveema, kathabar

Contents

1. Design and Implementation
    1.1. Server Class
    1.2. Server Simulator Class
    1.3. Load Balancer Class
    1.4. Client Class
    1.5. Cache Class
    1.6. Task Class
2. User guide to compiling and running the distributed application
3. Group work
4. Screenshots

Design and Implementation

We have the following main classes:

1. **Server Class** - Responsible for fetching the data, either from the client
2. **Database Class** - Responsible for querying the database. All the algorithms for the queries are here.
3. **ServerSimulator Class** - Creates 5 instances of server and adds them to RMI registry.
4. **LoadBalancer Class** - Balances the load between servers by sending client requests to servers with low load whenever possible.
5. **Client Class** - Responsible for simulating the clients. Fetches the server for the user request from load balancer and sends the requests to the server
6. **ClientRepository Class** - Acts as the interface between the Client and the Client data. It is responsible for fetching the data, either from the client cache, or the server, and returning it to the Client class.
7. **Cache Class -** It maintains the MusicProfile and UserProfile structures. Both Clients and Servers can use different cache objects to maintain cache. This class is not thread safe.
8. **Task Class -** It contains all the different parameters needed for the output files. Servers return results in one of the four classes (subclass of task class) for each type of query

1 . Server Class:

- services the client requests
- supports 4 different types of queries
- uses 2 threads. one executes the tasks and the other manages the waiting list.

Steps used to handle the given 4 queries:

1. if server side cache is enabled then query is checked against the cache
    a. if cache contains the result then the result is used
    b. if cache doesn't contain the result then the query is query performed on the dataset
2. if cache is disabled, then the dataset is queried to answer the query
    a. BufferedReader is used to read the dataset record by record
    b. for each record read, the dataset is parsed and the necessary data needed are stored in local variables
    c. The data is used for answering the query
    d. The cache is updated with the result
3. The result is constructed in the form of task object and sent back to the Client

Implementing the 4 queries:

1. function overloading is used to implement the queries
2. server interface contains only one method called "execute query"

3. for each query, the method will take a different object as input and return a different object as output. (input and output objects both extend the Task class. The purpose of extending the task class is to write to the output files easily)

simulating latency(80ms or 170ms):

1. the client sends the zoneId to the server as one of the parameters in the input (encapsulated as subclass of Task class)
2. The server based on the zoneId calls the sleep method for 80 or 170 ms

Thread Management:

1. ThreadPoolExecutor is used to create 2 threads (QueryExecutor and WaitListExecutor)
2. Whenever a server receives a new request, the server adds it to the QueryExecutor by calling the submit() method on the thread. The QueryExecutor executes the submitted tasks one by one.
3. Whenever load balancer requests the wait list details, the server sends this request to the WaitListExecutor thread. This thread checks the size of the QueryExecutor and returns it.

2 . Server Simulator Class:

- creates 5 instances of the server
- for each server the server simulator creates a cache object from cache class with a capacity of 100 music ids (if the -c flag is passed in the command line argument)
- The instances are added as remote objects to the rmi registry and binded using the names server1, …, server5

3 . Loadbalancer class:

- balances the load between the 5 servers by assigning low loaded server address to the client
- keeps tracks of the loads of all the servers
- server ids: 1,2,3,4,5 are used to identify 5 different servers. So it's easy to compare it against the zone ids.
- one object is created and added to rmi registry. So load balancer itself provides it's services as remote object. Client can access this remote object by fetching the stub from the rmi registry

Load Management:

1. It uses 2 HashMaps called ServerRequests and ServerWaitingLists
2. ServerRequests is keeps tracks of number of requests sent to each servers from the starting till the end
3. ServerWaitingLists contains the waiting listing count of each server and it's getting updated after every 10 new requests sent to the particular server

Choosing Server for client:

1. it's chooses a server which is in the same zone as the client and checks the load by checking the ServerWaitingLists structure
2. if the server is overloaded the check the neighbouring servers (left and right server) load (ServerWaitingLists structure)
3. if both neighbours are not overloaded then chooses the one with less load
4. if only one of the neighbours are not overloaded then choose that one
5. if both neighbours are overloaded then choose the current server
6. once a targeted server is chosen the the corresponding ServerRequests and ServerWaitingLists structures are updated
7. It returns the response (in the form of LoadBalancerResponse class) and it contains the stub of the chosen server and communication delay expected from client to the chosen server

4 . Client Class:

- parses input file
- creates the tasks and executes them
- calculates the average times
- prints the results in the output file

Task execution naive implementation:

1. Check if the cache is disabled
2. Parse the input file "input/naive_input.txt"
3. A client repository with a cache of null gets created
4. For each line a task of the type referring to the given methods gets created
5. Execute the execute method in the task specific object
6. This execute method calls the exact method (e.g. getTimesPlayed(musicId) which is defined in the cache object
7. The result will be stored in the task object
8. The client waits until all the requests are finished
9. The average turnover time, waiting time and execution time gets calculated in the client. The necessary data is stored in the task object for each request.
10. For each task a line with the method specific information is printed into the output file
11. In the end the average times are added

Task execution for client side cache

1. Check if the cache is enabled by checking for the flags "-c" and "-s" which mean that both client and server cache are enabled
2. Parse the input file "input/cached_input.txt"
3. A client repository with a cache gets created
4. The next steps are equal to 4 + 5 in the naive implementation
5. The difference when calling the method in the cache object in comparison to the naive implementation is that first of all the method checks if the given request can be handled with the data in the cache
6. If not, the method will be executed as in the naive implementation
7. The following steps are equal to the naive implementation

5 . Cache Class:

- Maintains MusicProfile and UserProfile based on the structure provided in the assignment description
- cache object is attached to both client and servers
- clients and servers can use the cache objects to answer the queries to preserve time
- It contains 6 methods. 2 methods to update the music profile and user profile each. Remaining 4 methods to perform the queries on the music profile and user profile.
- the difference between the client and server cache is the difference in the supported music id capacity

Music Profile:

- It contains music id and the artist id and used it as key for the musicProfiles object in the cache and also used in the user profile
- musicProfiles is a LinkedHashMap of size 100 or 250 that is attached with the cache object depending on whether it's with client or server. The key is music profile (music id, artist id). The value is number of times played.

UserProfile:

- it contains string userId to represent the user
- it contains the complex musicProfileMap structure of the form HashMap<String, HashMap<MusicProfile, Integer>>. The size of the musicProfileMap is 3.
- To store only recent 3 genres per user, separate genreOrder Deque is used and it will be adjusted whenever a new genre is added to the user profile.

Adding new Music Profile to Cache:

1. whenever the server/client encounters a getTimesPlayed query with new musicId, then the music profile in the cache will be updated
2. since it uses LinkedHashMap to store the information, automatically the oldest music profile will be deleted whenever it's full

Adding new User Profile to Cache:
The following steps are used to add new user profile to the cache.

1. check whether userId exists or not
    a. if it exists then check whether the genre exists for the user or not
        i. if it exists
            1. check whether the music profile for the genre is full or not
                a. if full then remove the oldest music profile and add the new one
                b. if not then add the new one
        ii. if it doesn't exist then
            1. check whether already 3 genres are present or not
                a. if it does then remove the oldest
                b. add the music profile and genre to the musicProfileMap
    b. if it doesn't exist then check whether userId Capacity is reached or not
        i. if it is full then remove the oldest entry
        ii. add the user profile

Hacks to handle different queries for the user:

1. genre called “userTimes” is used to store the musicId and the number of times user played. This is added to the user profile in the cache
2. genre called “topThreeMusicByUser” is used to store the top musics listened by the user
3. along with these two one more genre and it’s music profile also can be stored (top musics of a specific genre for user)
4. By using the above two keys to index the user profile in the cache, the cache can handle even all the 3 user related queries if it gets the input in the favorable order

6 . Task Class:

- abstract class for all tasks
- gets extended by task classes for the four different methods
- inherits the execute method which gets overwritten by the method task classes using polymorphism

subclasses:

- TimesPlayedTask, TimesPlayedByUserTask, TopThreeMusicByUserTask, TopArtistsByMusicGenreTask classes inherit the Task abstract
- Those 4 classes are designed to handle the 4 different queries given in the assignment
- The purpose of having 4 different classes were to reduce the overhead in the client when it passes different methods to the server. The corresponding classes store the arguments for the query. They also store the timings needed to calculate the waiting time, execution time and the turnaround time. The corresponding objects are passed to server and the server adds timings information to the same object.
- The main usage of the different tasks classes are it contains toString() methods which return the string needed to write it in the output files. So the toString() methods perform the formatting.

User guide to compiling and running the distributed application

We use the terminal to run our application. For each component we use a separate terminal window which is navigated to the src folder.

steps:
1) The first step is to compile the java files.

```
javac *.java
```

2) run the rmiregistry

```
rmiregistry
```

3) run the serverSimulator // -c for enabling cache in the server

```
option 1 ) java ServerSimulator -c
option 2) java ServerSimulator
```

3) run the LoadBalancer

```
java LoadBalancer
```

3) run the client // use -c flag to enable client side cache. use -s flag to denote that server is added with cache (it's a hack to tell client that server contains cache to print the output)

```
option 1) java Client -s -c
option 2) java Client -s
option 3) java Client -c
option 4) java Client
```

Group work

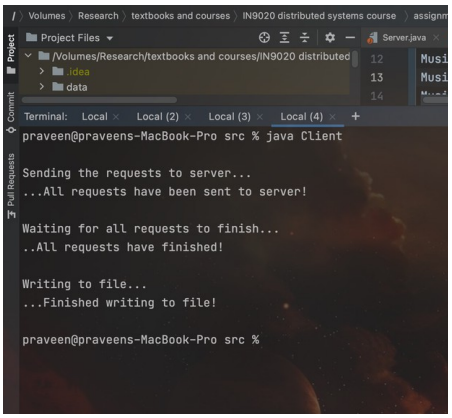
We met three times within the assignment period to discuss the general architecture and answer questions if any occurred within the time working on our own.
On the first meeting we discussed and designed the architecture of our naive implementation and set up a basic hello world rmi example, so everyone has a working environment to start working on. After that we split up the work. One person did the Server implementation, one person the loadbalancer and one person the client.
After that we met again to merge the three component's code and solved merging issues. We discussed the work that has to be done before working on the cache and split it up in similar workload amounts between us three again. Additional two people started working on the cache structure, while one person fixed a bigger problem with the threads.
After that we met again to finalize the cache structure and implement it.

Screenshots

The screenshots are also available in the documentation folder as well as this document.

Client under execution:

1. Without caching



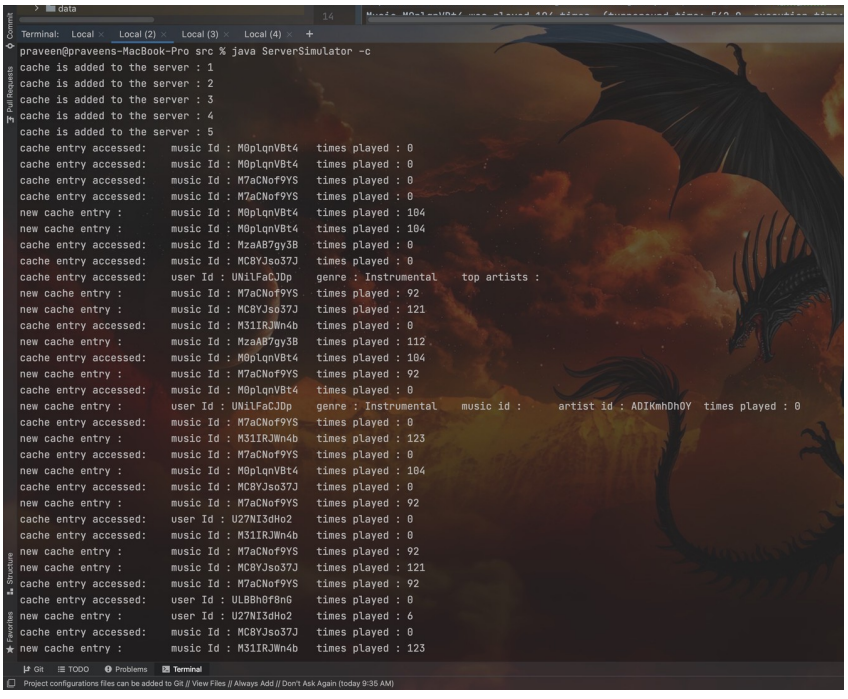
2. With caching:

```
cache entry accessed:    music Id : M88BUa8CMM   times played : 0
TimesPlayedTask : music id : M88BUa8CMM  count : 0
cache entry accessed:    music Id : MbpGLqUmJL   times played : 0
TimesPlayedTask : music id : MbpGLqUmJL  count : 0
cache entry accessed:    music Id : M0plqnVBt4   times played : 0
TimesPlayedTask : music id : M0plqnVBt4  count : 0
cache entry accessed:    music Id : MbpGLqUmJL   times played : 0
TimesPlayedTask : music id : MbpGLqUmJL  count : 0
cache entry accessed:    music Id : MdNa6XTLrc   times played : 0
cache entry accessed:    music Id : M0plqnVBt4   times played : 0
TimesPlayedTask : music id : M0plqnVBt4  count : 0
cache entry accessed:    music Id : MRrvdie1Y8   times played : 0
TimesPlayedTask : music id : MdNa6XTLrc  count : 0
cache entry accessed:    music Id : MDG9PpEMU4   times played : 0
TimesPlayedTask : music id : MRrvdie1Y8  count : 0
cache entry accessed:    music Id : MF4b3OBnCf   times played : 0
TimesPlayedTask : music id : MDG9PpEMU4  count : 0
TimesPlayedTask : music id : MF4b3OBnCf  count : 0
new cache entry :        music Id : M8giK3yRJq   times played : 95
new cache entry :        music Id : MmAp67iFLc   times played : 111
new cache entry :        music Id : MmAp67iFLc   times played : 111
new cache entry :        music Id : MC8YJso37J   times played : 121
new cache entry :        music Id : MbpGLqUmJL   times played : 93
new cache entry :        music Id : MadldTEN8S   times played : 107
new cache entry :        music Id : MHZBMyFxWV   times played : 152
new cache entry :        music Id : MrSx5dbQrm   times played : 109
new cache entry :        music Id : MrxHdptXup   times played : 124
new cache entry :        music Id : MPBXdzYFmt   times played : 92
new cache entry :        music Id : M31IRJWn4b   times played : 123
new cache entry :        music Id : MgVv5S8y7W   times played : 119
new cache entry :        music Id : M31IRJWn4b   times played : 123
new cache entry :        music Id : Mf4rI55g8j   times played : 127
new cache entry :        music Id : MG0KDHIBzo   times played : 114
new cache entry :        music Id : MIOsWAILVR   times played : 110
new cache entry :        music Id : MPBXdzYFmt   times played : 92
new cache entry :        music Id : MdXYfKPxaI   times played : 97
new cache entry :        music Id : MHZBMyFxWV   times played : 152
new cache entry :        music Id : MTRMfUKEH2   times played : 128
```

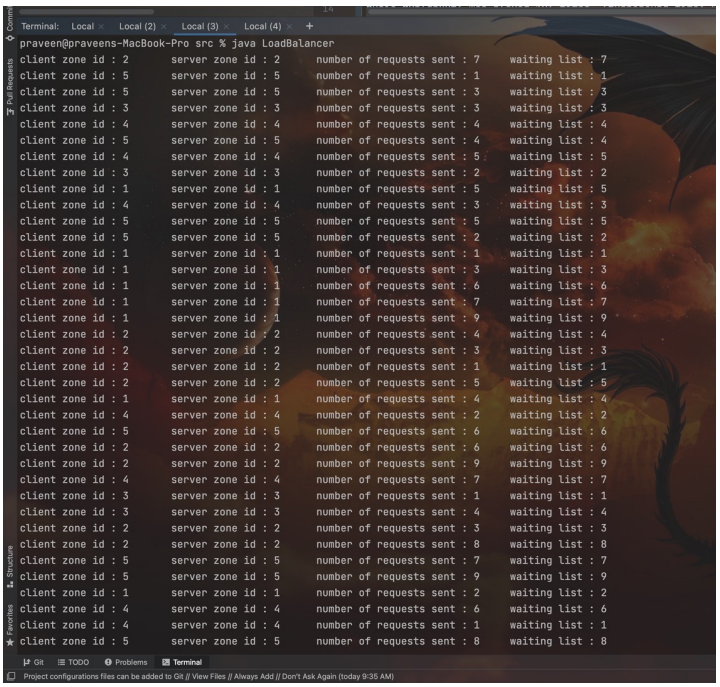
```
new cache entry :        music Id : M0plqnVBt4   times played : 104
new cache entry :        music Id : MRrvdie1Y8   times played : 191
new cache entry :        music Id : MRrvdie1Y8   times played : 191
new cache entry :        music Id : MbpGLqUmJL   times played : 93
new cache entry :        music Id : MRrvdie1Y8   times played : 191
new cache entry :        music Id : MbpGLqUmJL   times played : 93
new cache entry :        music Id : M88BUa8CMM   times played : 161
new cache entry :        music Id : MaaVEtbteM   times played : 107
new cache entry :        music Id : MaaVEtbteM   times played : 107
new cache entry :        music Id : MrSx5dbQrm   times played : 109
AKaDk1WKOQ
APqkESNTY7
null
new cache entry :        music Id : MrxHdptXup   times played : 124
new cache entry :        music Id : MgVv5S8y7W   times played : 119
new cache entry :        music Id : MgVv5S8y7W   times played : 119
new cache entry :        music Id : MC8YJso37J   times played : 121
new cache entry :        music Id : MzaAB7gy3B   times played : 112
new cache entry :        music Id : MIOsWAILVR   times played : 110
new cache entry :        music Id : MG0KDHIBzo   times played : 114
new cache entry :        music Id : MC8YJso37J   times played : 121
new cache entry :        music Id : MZu6bT20PV   times played : 100
new cache entry :        music Id : MadldTEN8S   times played : 107
ADIKmhDhOY
AFzoPnPyWA
null
new cache entry :        music Id : Mo72ZF2Kal   times played : 122
AKaDk1WKOQ
null
APqkESNTY7
new cache entry :        music Id : MPBXdzYFmt   times played : 92
new cache entry :        music Id : MZu6bT20PV   times played : 100
new cache entry :        music Id : MdNa6XTLrc   times played : 150
new cache entry :        music Id : MDKKd0ES1j   times played : 148
..All requests have finished!

Writing to file...
...Finished writing to file!
```

Server under execution:



LoadBalancer under execution:



The output files can be found at src/output.