

**COMP6231- Distributed System Design**

**Assigment-2**

Distributed Player Status System (DPSS) using Java IDL

**Date:** 28 June 2020.

**Recipient:** Prof. Mohammed Taleb

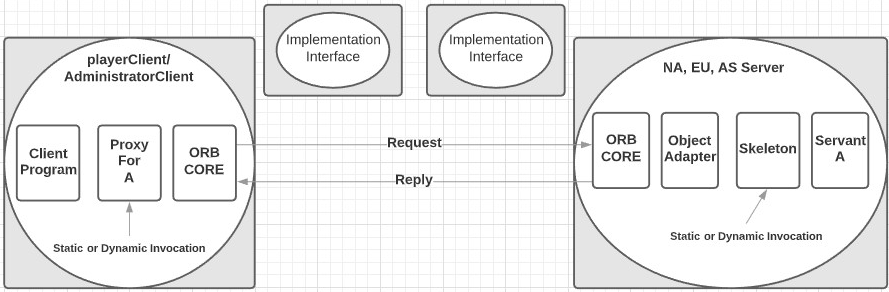
**Student Name:** Raj Mistry (40119206)

* **Introduction:**

The main objective of this assignment is to design and implement a Distributed Player Status System which manages player’s status across multiple game servers using Java IDL & CORBA. CORBA is one of the middleware solutions based on distributed object that offers simple way to call different methods on server machines to perform various operation and returning objects over network and provide network transparency. In addition to this we have to make our application synchronized with the use of multi-threading in order to handle concurrent requests with ease. User Datagram Protocol is used by each server to communicate with other servers, sending and receiving requests to perform inter-server operations.

* **CORBA Architecture:**

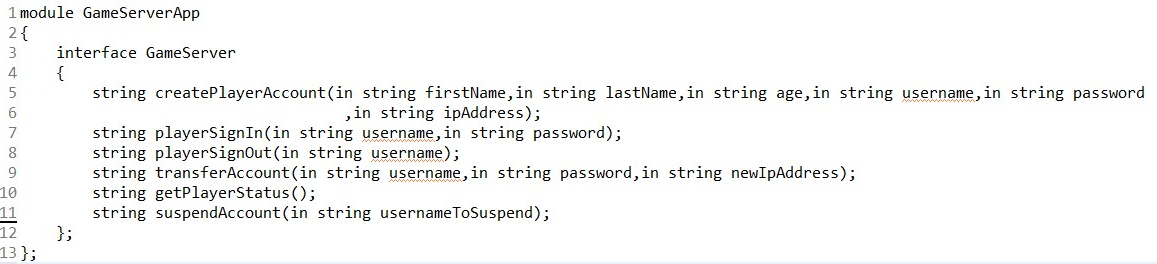
In this assignment the architecture used is CORBA – Common Request Broker Architecture. It is not two-tier or three-tier architecture, it is middleware which is designed to allow the communication between various devices that are deployed in different platforms or different OS. As middleware it receives the requests from the client and transfer it to particular server object and then it marshals the exception or response from server and send it back to client. It uses the object-oriented model to provide its benefits like data encapsulation, abstraction, etc to provide transparency. For using object-oriented model, it is not required that the system using CORBA need to be Object-oriented; it enables the communication between systems irrespective of their programming language as well. It allows client to invoke server methods both locally and remotely (statically & dynamically).



**\* CORBA Architecture \***

* **IDL:**

IDL stands for Interface Definition Language or Interface Description Language. It’s a specification language which is used to describe the program’s Interface which is not dependent on any programming language. It defines name and methods that client can request. It enables the communication between client and server programs which may or may not be written in same programming language. IDL commonly used in RPC Software where at the both end systems may run on different OS and use different programming language. It provides bridge between these systems.



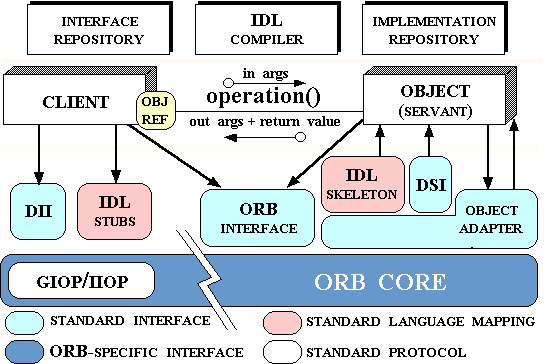
**\* IDL Code Snippet \***

* **ORB:**

The Object Request Broker is a middleware that uses CORBA specification. ORB manages all the details involved in sending the request form client to object and sending the response to its destination. ORB also manages the transformation of data structure of the process to the raw byte sequence and vise-versa, which is required to transmit it over the network. This is called marshalling or serialization. Other than this ORB also provides features like distributed transactions, directory services or real- time scheduling. The ORB is also responsible to looks after the Interface Repository, an OMG (Object Management Group)-standardized distributed database containing OMG IDL interface definitions.

On client side, ORB provides interface definition and performs method invocation using dynamic invocation interface (DII).

On the server side, the ORB de-activates inactive objects, and re-activates them whenever a request comes in. CORBA supports a number of activation patterns, so that different object or component types can activate and de-activate in the way that uses resources best.

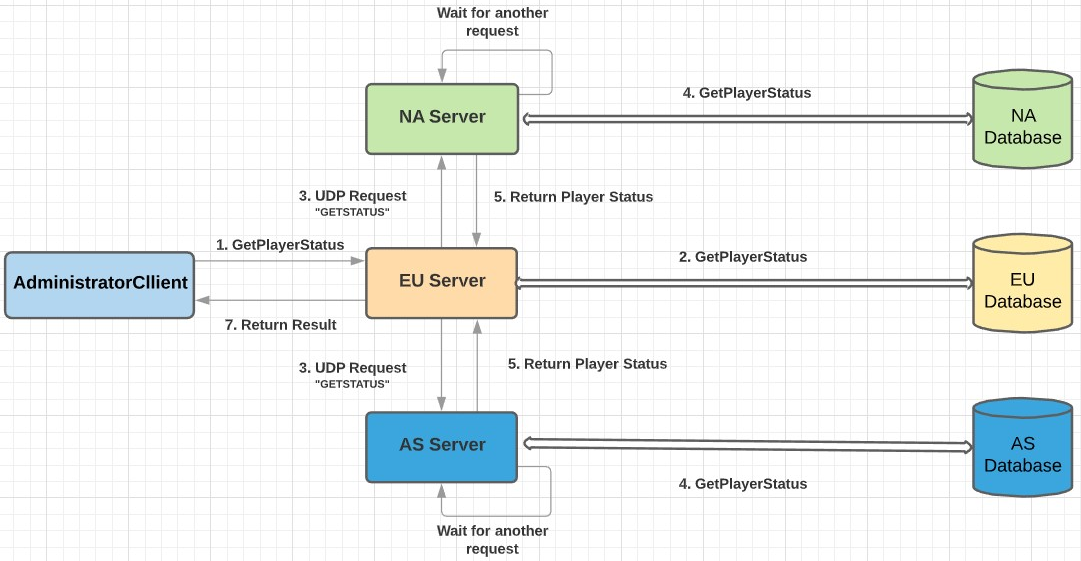


**\* ORB Architecture \***

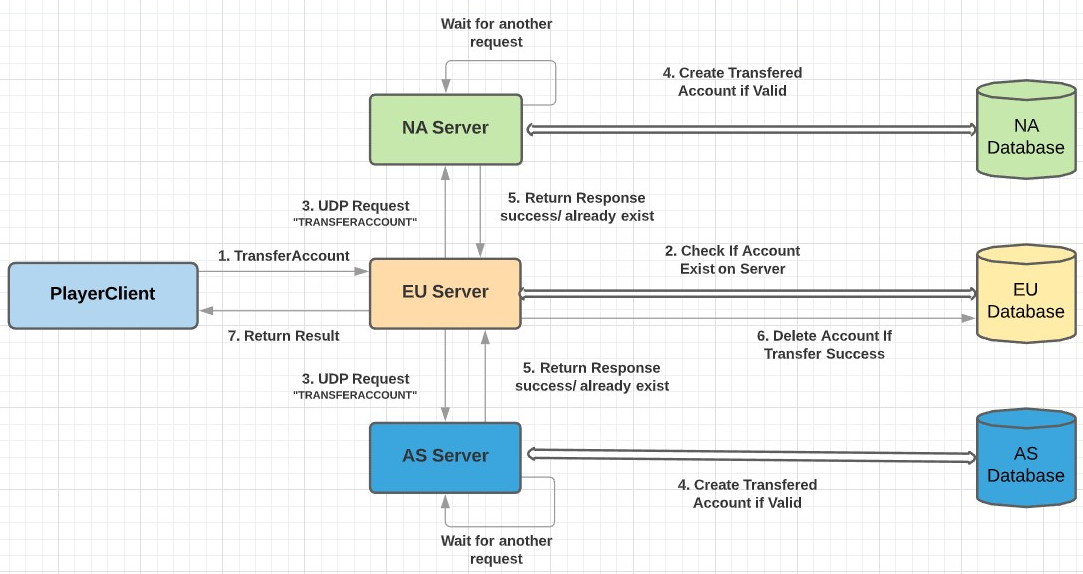
* **UDP:**

The User datagram Protocol is a transport layer protocol. It is connectionless and unreliable protocol. To communicate using UDP it is not required to establish connection prior to data transfer. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The Datagram Packet and Datagram Socket classes in the java.net package implement datagram communication using UDP.

* In this assignment I have implemented UDP for communication among three servers (NA, EU and AS) in order to perform operations getPlayerStatus() and transferAccount(). Each server Creates its own UDP thread when server starts.
* In getPlayerStatus () the current server communicates with other servers using UDP request “GETSTATUS” to fetch the player accounts status on corresponding server and returns server specific online and offline player accounts count. (In this UDP request will be sent to rest of the 2 servers)



* In transferAccount() checks the HashMap for the required player account. If found, the server transfers the account to specified remote server by sending UDP request “TRANSFERACCOUNT” and after successfully transferring the record to remote server, it will remove the record from its storage and then server informs the client that the player account transfer is successful.(In this UDP request will be sent to only 1 server where we want to transfer Player Account[Destination Server] e.g. To transfer Account from EU to AS, UDP Request will be sent to AS only not to NA])



* **Multithreading and Synchronization:**

Thread is a lightweight process which has its own stack. When Multiple threads try to access the same resources and produces wrong result because of data corruption. This happens because while one thread is accessing resource same time other thread interferes and access it and now both perform operation on that old value and write their result which may produce wrong result.

Synchronization is a procedure in which is used to avoid interference with the other threads and memory consistent errors. The keyword synchronized is used to implement Competition Synchronization which means while one thread is executing the critical region where shared resource is used all the other threads will wait until its execution of critical region finished.

There are Two types of synchronizations used in java:

* Method Synchronization
* Block Synchronization.

***Synchronization Used in this Assignment:***

* Block level Synchronization:
  + A specific part or block of code that is needed to be synchronized is put under its own specific scope.
* Method level Synchronization
  + A specific methods or functions in java is synchronized using synchronize keyword before the method name.
* **Data Structure and Concurrency:**

Here in this assignment in order to improve concurrency I have used ConcurrentHashMap to as the storage for Player Accounts as it is an enhancement of HashMap, as we know that while dealing with threads in our application simple HashMap is not good choice because performance wise HashMap is not up to the mark. ConcurrentHashMap class is thread safe and it is good choice when we need high concurrency in our application. It does not throw a “ConcurrentModificationException” if one thread tries to modify it while another is iterating over it.

**Syntax:**

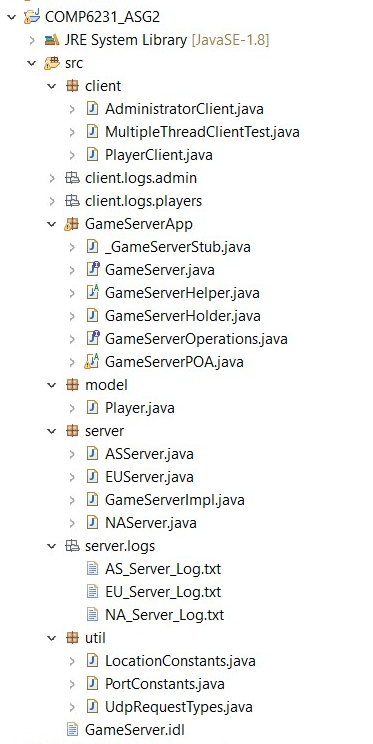
ConcurrentHashMap<String, List<Object>> accounts = new ConcurrentHashMap<String, List<Player>>();

Where Player is the Model class which stores details of player account in its object.

The Fine-grained locking is used to put intrinsic lock while doing parallel access to data resources so that data is accessed as required without infecting the data. This increases the efficiency and maximizes the concurrency to perform concurrent operations.

Concurrency is implemented by using the synchronized(){}-block in java which puts temporary lock on the object or list passed to it as parameter in this implementation.

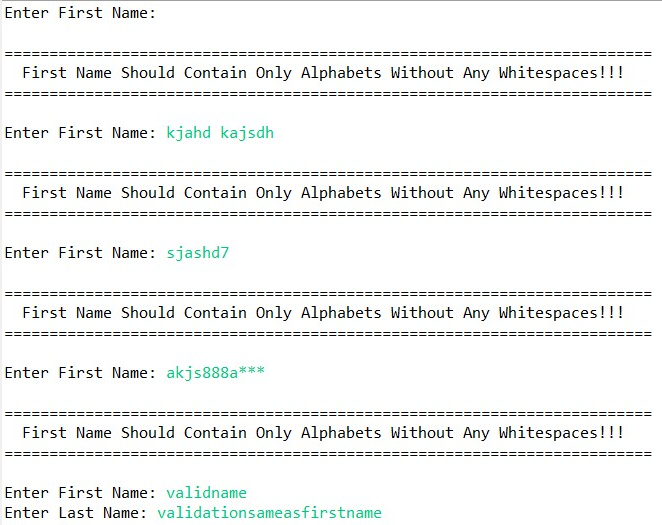
* **Code Structure:**

****

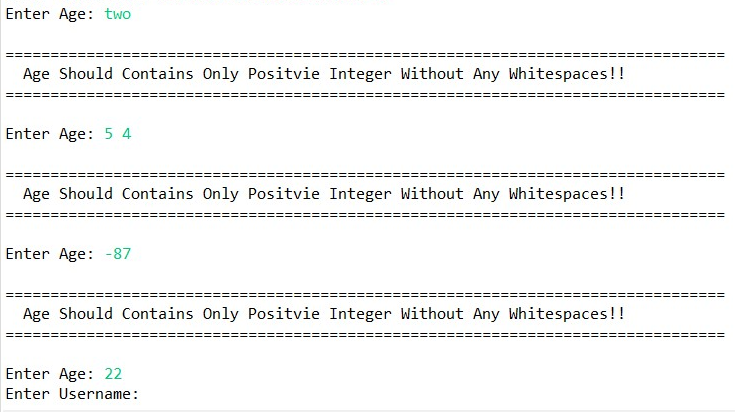
* **Test Cases:**

1. **Validations (All attributes must be entered & can’t be empty)**

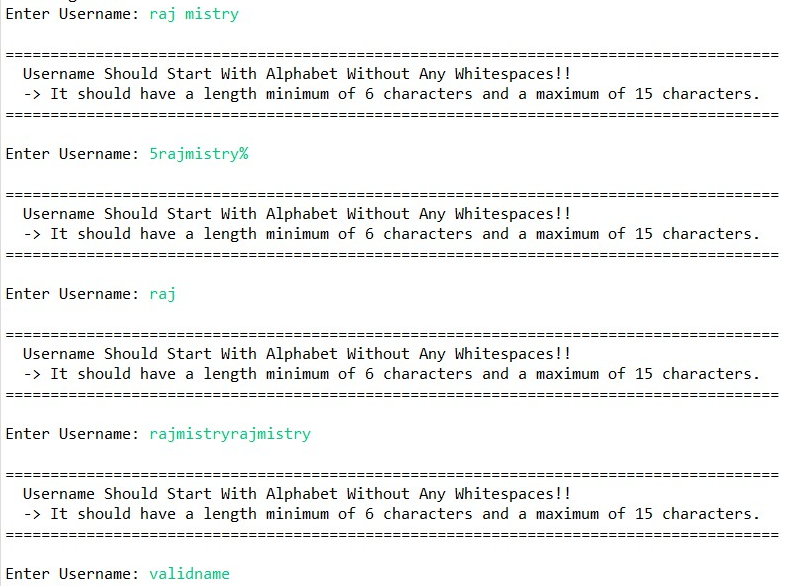
* **First Name** and **Last Name** Should Contain Only Alphabets without any Whitespaces.



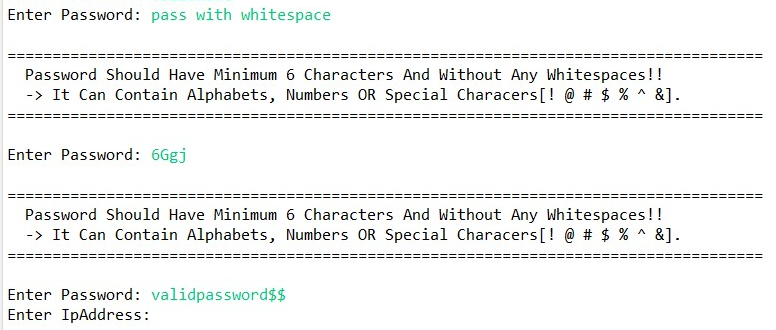
* **Age** must be Positive Integer without any whitespaces.



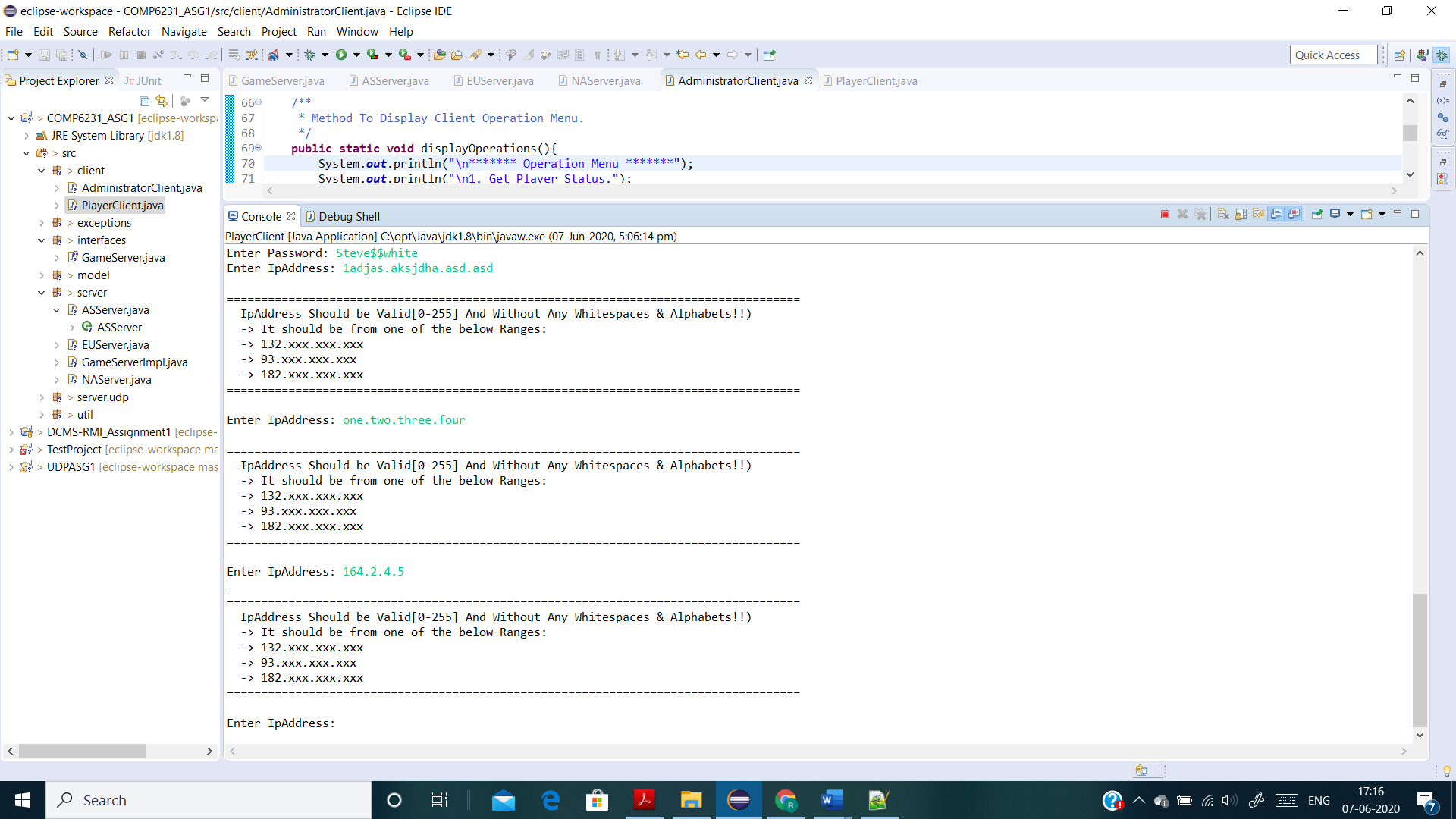
* **Username** must be start with alphabet and should not contain any whitespaces and length must be minimum of 6 and maximum of 15 characters.



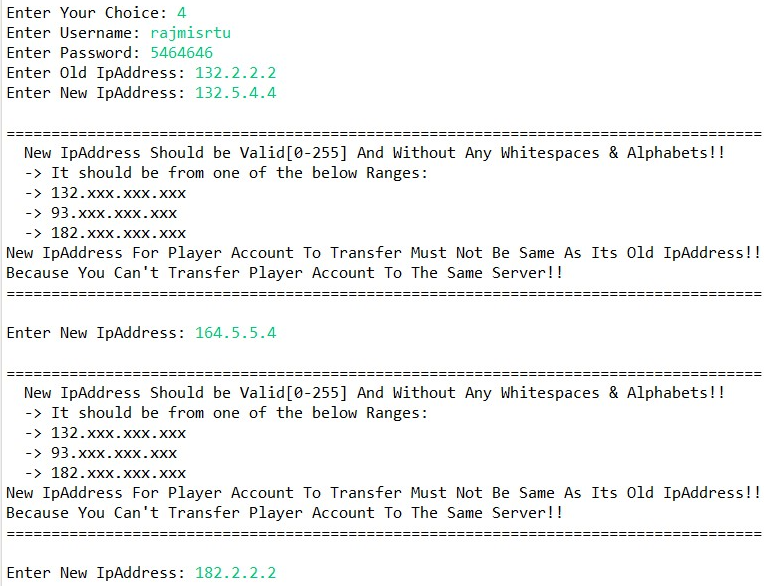
* **Password** must not contain any whitespaces and should have minimum length of 6 characters.



* **IP Address** must be entered as per valid IPv4 format & must be entered from one of the 3 ranges (132.xxx.xxx.xxx, 93.xxx.xxx.xxx, 182.xxx.xxx.xxx)



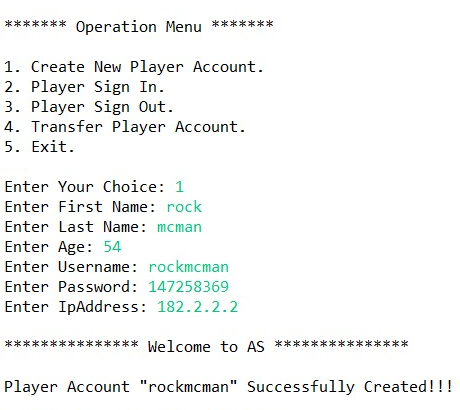
* **New IP Address for transfer Account** must be different from old Ip Address as Account cannot be transfer to same server & it must be entered as per valid IPv4 format & must be entered from one of the 3 ranges (132.xxx.xxx.xxx, 93.xxx.xxx.xxx, 182.xxx.xxx.xxx).



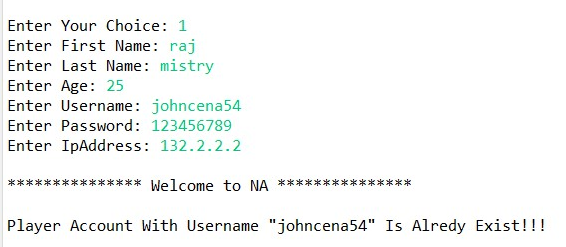
* **Username and Password for Administrator** must be “Admin” without any whitespaces and must be exact same as shown (Case Sensitive).



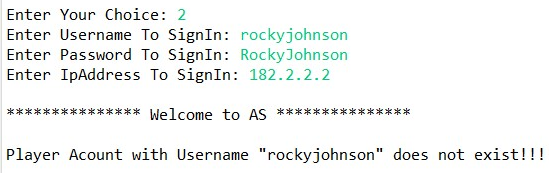
1. **Create New Player Account (Successfully Created New Player Account)**

****

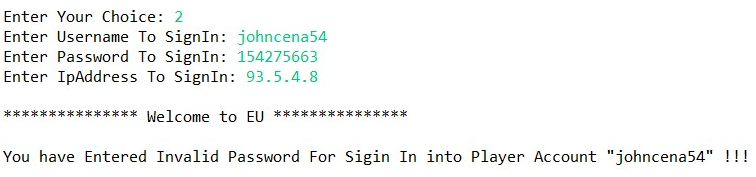
1. **Create New Player Account (Unsuccessful – Player Account with same username Already Exist)**

****

1. **Player Sign In (Unsuccessful - Player Account in which trying to Sign In Not Exist)**

****

1. **Player Sign In (Unsuccessful - Wrong Password Entered For Player Account to Sign In)**

****

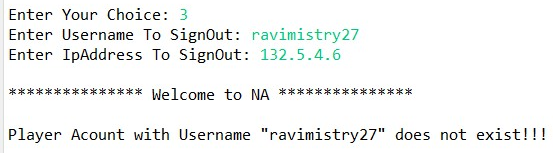
1. **Player Sign In (Successfully Signed in into Player Account)**

****

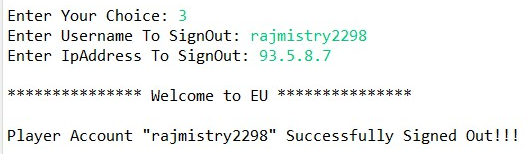
1. **Player Sign In (Trying to sign in into Player Account which is Already Signed In)**

****

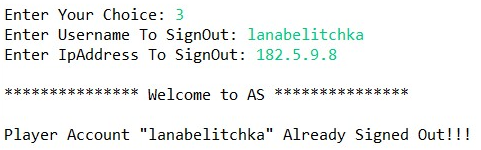
1. **Player Sign Out (Unsuccessful - Player Account from which trying to Sign Out Not Exist)**

****

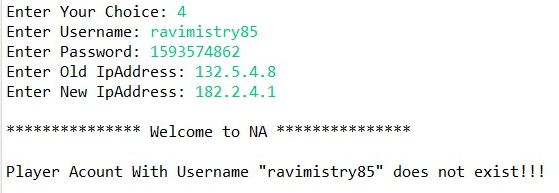
1. **Player Sign Out (Player Account Successfully Signed Out)**

****

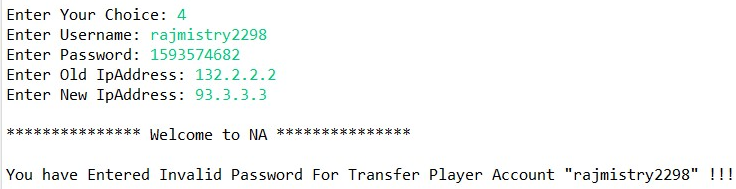
1. **Player Sign Out (Player Account Trying To Sign Out Which Is Already Signed Out)**

****

1. **Transfer Player Account (Unsuccessful – Player Account which client trying to transfer does not exist on server)**

****

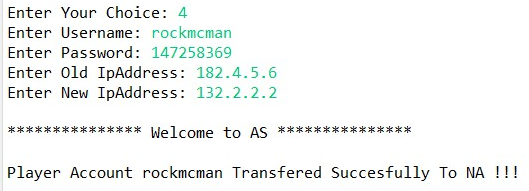
1. **Transfer Player Account (Unsuccessful – Player Account which client trying to transfer is exist on server but wrong password is entered for Account)**

****

1. **Transfer Player Account (Unsuccessful – Player Account which client trying to transfer, Player Account with same username already exist on destination server)**

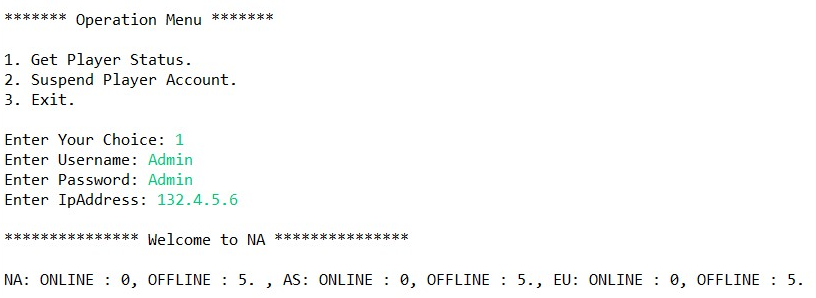
****

1. **Transfer Player Account (Successful – Player account exist and valid password and account with same username not exist on destination server.)**

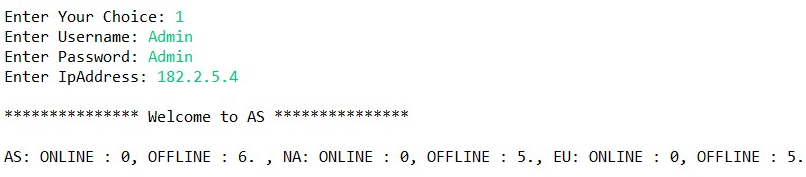
****

1. **Administrator Get Player Status.**

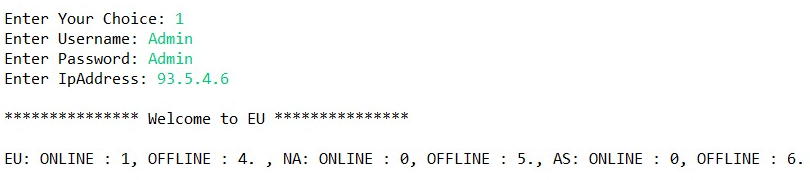
**Initially each server has 5 Player Accounts:**

****

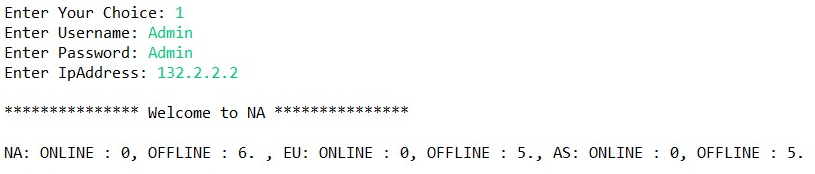
**After Creating One Player Account On Asia Server by playerClient:**

****

**After One Player Account Signed In at Europe Server by playerClient:**

****

**After Transferring 1 Player Account from Asia Server to North-America Server.**

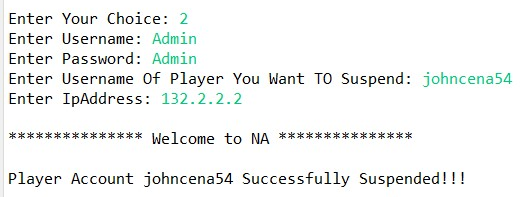
****

1. **Suspend Player Account.**

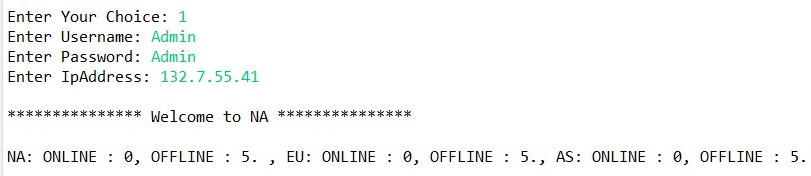
**Unsuccessful – Player Account with entered username not exist on server.**

****

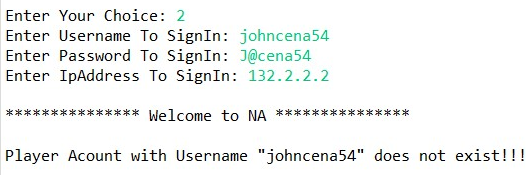
**Suspend Player Account successfully.**

****

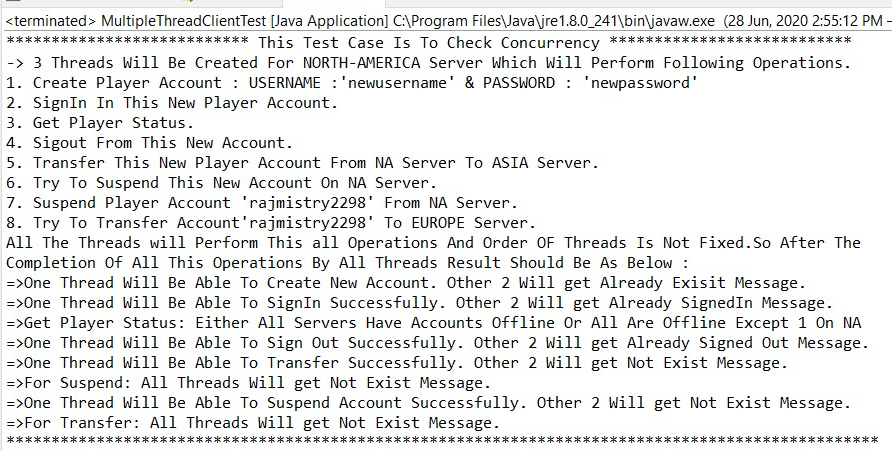
**Player Status after Suspending One Account from NA server.**

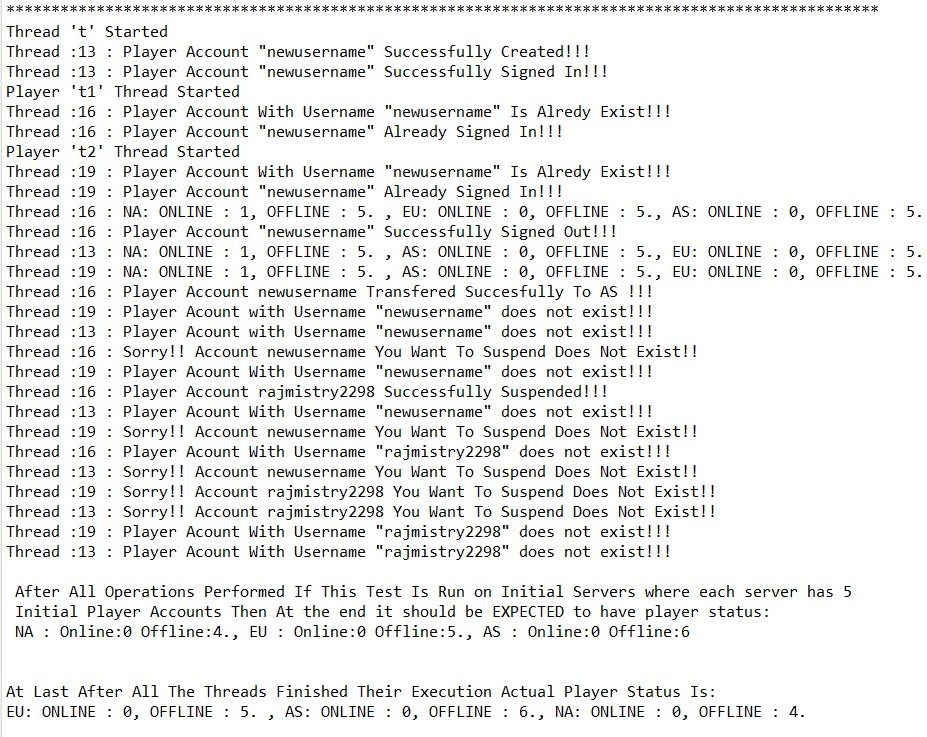
****

**Try to sign in into player account which was suspended by Admin.**

****

1. **Multithreaded Test For Checking Concurrency and Atomicity for Transfer Account And Suspend Account.**

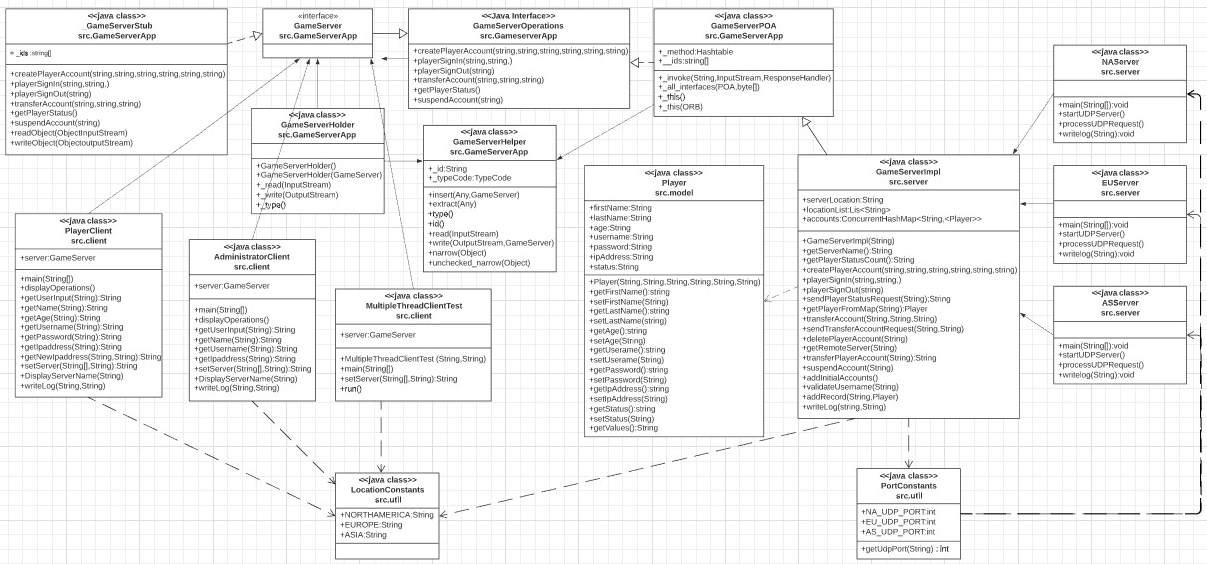
****

****

* **Challenges (Most Important / Difficult Part):**

1. CORBA implementation.
2. Implementation of Threads for UDP.
3. Implementing synchronization among shared data and avoiding deadlocks.
4. UDP implementation of getPlayerStatus() and transferAccount().

* **Class Diagram:**

****

* Here GameServerApp is Created by compiling GameServer.idl file. Folder contains all the autogenerated files: GameServerPOA, GameServerOperation, GameServerHolder, GameServerHelper, GameServerstub and GameServer interface. These all file are used in implementation.
* For Creating Server classes and Client Classes GameServer and GameServerHelper are used from GameServerApp.
* For GameServerImpl it inherits GameServerPOA from GameServerApp.
* First in all 3 servers the server side implementation of CORBA is done which includes Create & Initialize The ORB, Get Reference To rootPOA & Activate The POAManager, Create Servant & Register It With The ORB, Get Object Reference From The Servant, Cast The Reference To A CORBA Reference, NameService Invokes The Transient Name Service, Bind the Object Reference In Naming and then run the ORB. With this the UDP Thread is also started in each server which will start the UDP Server for that particular Server and will wait for request.
* PlayerClient and AdministratorClient runs the infinite while loop which asks user what operation user wants to perform and based o the input it will take required inputs from user along with the IpAddress. This IpAddress is used to determine on which server user want to perform operation and based on that the client side CORBA connection to particular server is done and on that server the operation is performed and get the response and displays it to client and again loop is continued to ask user about operation choice. When user enters choice of exit the client program terminates.
* In GameSeverImpl all the methods of GameServer interface are implemented and data is managed in ConcurrentHashMap. each server has its own copy of data. In this proper Synchronization both method level and block level is used along with countdownlatch to maintain consistency of data and manage & increase concurrency and atomicity.
* **References:**
* <http://www.ejbtutorial.com/corba/tutorial-for-corba-hello-world-using-java>
* http://bedford-computing.co.uk/learning/wp-content/uploads/2016/03/george-coulouris-distributed-systems-concepts-and-design-5th-edition.pdf
* <https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture>
* <https://en.wikipedia.org/wiki/Interface_description_language>
* <https://en.wikipedia.org/wiki/Object_request_broker>
* <https://www.corba.org/orb_basics.htm>
* <https://research.cs.queensu.ca/home/xiao/DS/node5.html>
* <https://www.geeksforgeeks.org/countdownlatch-in-java/>