COMP 416

**Project #1: Protocol Design and Development of a Networked Card Game**

GROUP #14: Bulut Bulgu, Deniz Karadayı, Özgür Budak

DATE: 23th February, 2020

yatay çizgi

# 

# Master Side

**Client Side**

# Follower Side

# MongoDB

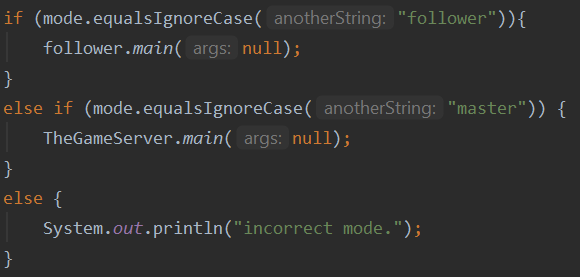
# Task Division

# Config File

# 

# Main Frame of the Project

For the game, there are two predefined essential parties which are game server branching into “master” and “follower”, and client as players. On the game server side, the *main* program is run in the beginning. The user is asked to specify whether it wants to be a master, or a follower and a the main of TheGameServer or follower class is called accordingly. On the client side, there is no branching and user directly connected to Client and open a Client object.



**Game Logic**

TheGameServer is the supplier of the game where Client is the player. When there are two players, a game will be assigned to them and they receive the half deck of shuffled cards. They both expected to send single cards for each round, and the one sent the valuable card will win the round. The goal for a client is to win as many rounds as possible to collect higher points. When both of the clients send the all cards they have, the result will be announced. Follower is the backup server for the main server in case it may fail.

# Master Side (*TheGameServer.java*)

If master is selected, server sockets are created (one for data, one for command) and command port begins listening for incoming connections. When something connects to the master, they announce themselves by sending a string containing that whether they are a follower or a client. If the connected party announces that it is a follower, data socket is also used and input and output streams are created, and these streams are passed into the *followerHandler* thread which will be started to take over the responsibility of communicating with this follower. Likewise, if the connected party announces it is a client, we add its socket to our list of player sockets. If there is a corresponding player to accompany, we create a *SingleGameServer thread* using the sockets of both players. Then the *SingleGameServer* handles the game logic and both players.In other words, *TheGameServer* manages the traffic of connected sockets and directs them accordingly.

*Management of Multiple Games*

Coming player sockets are being added to a list of player sockets. For each new even number of sockets, a new *SingleGameServer* thread is being started as shown in the code below.

****

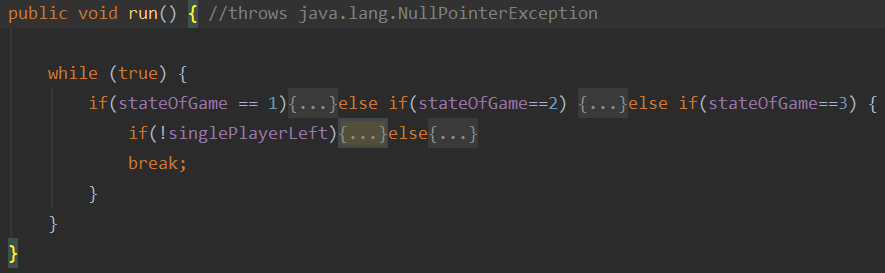
Implementation of the Game

The *SingleGameServer* class includes another class called *PlayerThread*. When a game is assigned for a pair of clients, *SingleGameServer* starts a pair of *PlayerThread* as well which separately takes the socket of their interest of clients as parameter. Each *PlayerThread* object provides the communication between *SingleGameServer* and *Client*. They manage the game client-wisely, follow the messages coming from the clients and make sure that it fits the flow of the game. When it is needed, it interferes to handle an issue before it is being transmitted to the *SingleGameServer*. *SingleGameServer* merges the both replies coming from the clients through PlayerThread objects, and ensures the game is processing properly. It keeps the card data, computes the round results and gives feedback as it gets synchronized messages from both of the *PlayerThread* objects.

SingleGameServer.java Class

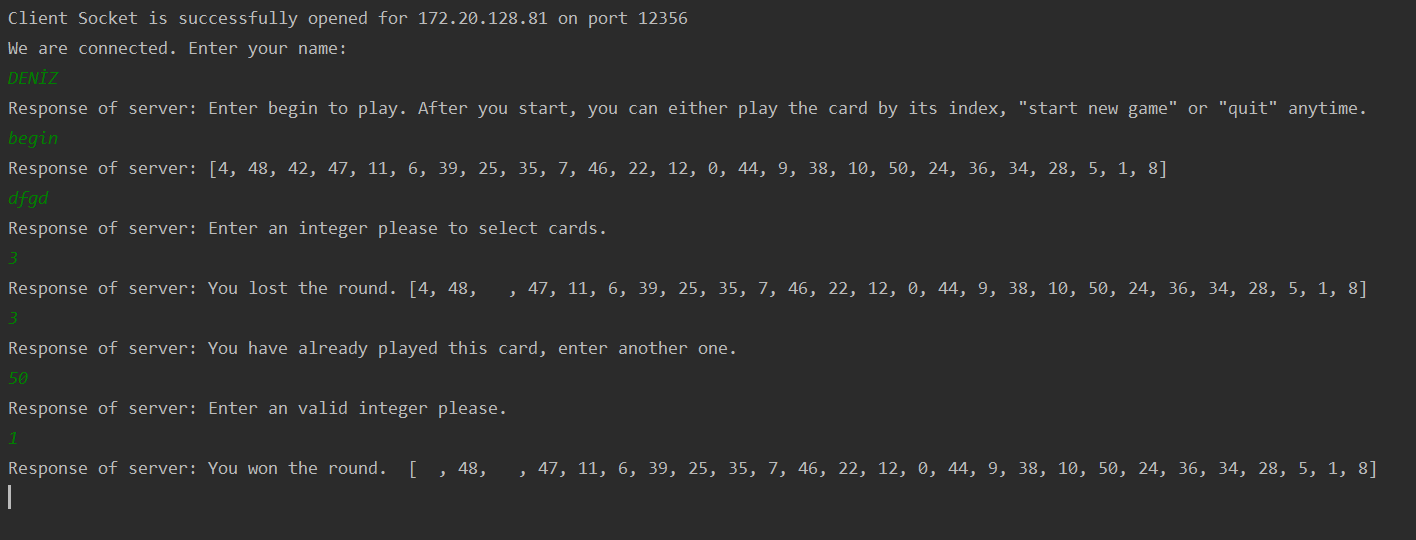
This class manages the game by following the both *PlayerThread* objects for at all states of the game. It tooks all the information of the players from the related PlayerThread objects and sends responses again via it all the time. The states of the game is listed below:

* State of Game 1: Players have entered their names, “begin” command is being waited.
* State of Game 2: Players have entered “begin”, SingleGameServer starts and handles the game.
* State of Game 3: Players have played their last cards or one of the players left the game, SingleGameServer waits until both of them exist.

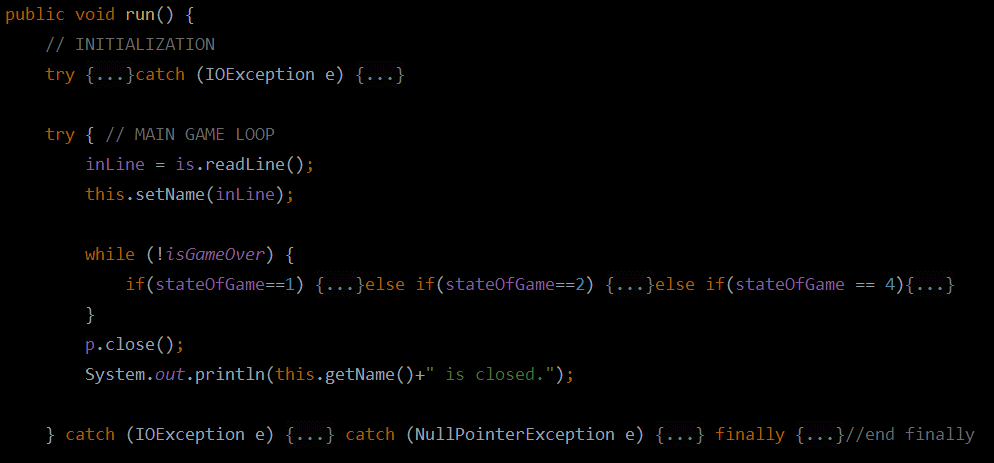
Initially, if both of the players entered their names, SingleGameServer waits inside of a loop for each one to send “begin” to start the game. In this state, always also checks if the both of the clients are still connected (to handle game left cases). At the second state, SingleGameServer sends the shuffled cards to players and waits in a loop until it receives both of the played cards. Once both cards are played, it computes the result of the game updates the card lists and sends them back to players with round results as well. If the state of the game becomes 3, either game is over after the last cards are played or a player left the game. According to the case, SingleGameServer ensures that each of players left the game properly, and their PlayerThreads get the related parameters telling the status of the game which are int:*stateOfGame* and boolean:*isGameOver.* The simplified structure of the run method of the SingleGameServer is shown below.

PlayerThread.java Class

PlayerThread acts as the main server communicating the Client socket, it only deals with its Client using TCP protocol. It sets an input and output streams to get and send streams with Client and directly connects it. The first message sent by the client is set as the name of the PlayerThread which will be accessible to SingleGameServer. The messages sent from the Client are kept as strings and tried to be matched with the expected strings of each state. In each state, there is a possibility that the Client wants to quit the game, so the case of quit requests is always considered and handled. Otherwise, Client is always expected to send messages related to the state such as “begin” command or index of played cards. For those cases, corrections are being asked Client to be done before the message is taken by the SingleGameServer as processable data. An example of correction requests sent from PlayerThread is illustrated in the following communication.



If a single player leaves, the PlayerThread of the continuing client will not know it until SingleGameServer warns it. Therefore, there is a fourth state description for PlayerThread which only can be set by SingleGameServer and indicates the left of the other player. For PlayerThread to know that game is over, its isGameOver boolean must be set true by SingleGameServer. Normally, when the last card is sent, PlayerThread sets its state to 3 but does nothing until isGameOver is true. In other words, PlayerThread has four states while SingleGameServer has three, and the third state is not handled by PlayerThread, instead, turns within the main while loop for SingleGameServer to make game ends. The simplified structure of the run method of PlayerThread is shown below.

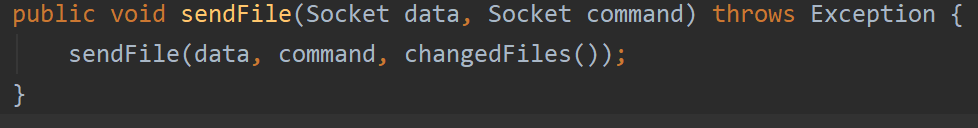
When a single player leaves the game, it is announced to the other one with the current scores by SingleGameServer. The remainder client is also asked to play a new game or quit. The actions of starting a new game or quitting is not handled by the game servers. Gamer servers only ensures that clients securely exit from game. Rest is handled by the Client side.

FollowerHandler.java Class

Any time a follower connects to the master, a FollowerHandler thread is assigned to that follower. FollowerHandler sends .json gamestate files to the follower, per follower's request. The follower checks the integrity of those files and may request the same files if there was a transmission error. The first thing the followerHandler does is to write the unique followerNo of the follower and then it keeps checking for messages from the follower. If a send command is received, it calls the send file method.

The sendFile method uses the changedFiles method to check the Master Files directory for any changed files. listFiles method is used on the directory to get a list of files in the directory. This check is done with the help of the fileHashMap. As its name suggests, each file is put in the hashmap with its lastModifiedDate. If the file doesn’t exist in the hashmap, or if its lastModified doesn’t match (meaning there was an update on the file), the file is added to the changedFiles arraylist to be returned from the method.

The sendFile method is an overloaded method. In its first version, it calls itself with the output of the changedFiles() method:



In the general case, it receives the changedFiles array (this case is called when there was a transmission error). Two sockets are used (command, data) and their corresponding input and output streams are created. The follower is sent the number of changed files. And then each file is sent. After a file is sent, the hash of the file is sent. After all files are sent, a final message is sent by the follower. If we receive the CONSISTENCY\_CHECK\_PASSED message we're done. Otherwise, we receive the RETRANSMIT message and get the number of corrupted files and their names. We call the sendFile method again using those files as the changedFiles arraylist.

# 

# Client Side (*Client.java*)

This is the class managing the client’s communication with the game servers. It initializes a socket for itself to transmit messages and trying to connect the ip of the main game server initialized my TheGameServer. Its name is automatically asked not by the game server but by itself. Until a second player is found for itself it receives nothing. Once a game is assigned, the PlayerThread sends it a message that asks to begin the game while it takes the initial stream sent by the Client as its name. Unlike game server classes, Client class uses its objects within itself which makes it more flexible in terms of repeating itself. In the main method, it creates a Client object, the rest of the code consisting of the method calls to manage the object. The method runTheClient handles all the work needed to be done on the object which are socket defining, connecting, TCP message loops etc. Until it gets a “quit” or “start new game” message, it keeps the send/receive message loops. Once it scans an exit message, it calls the exit method to close the socket and I/O streams. In the exit method, after closing thoses, it is either quit or create a new Client object to start a new game. It calls a runTheClient method for its new object to do the rest of the same work such as socket defining, connections, message loops etc. Therefore, Client class is kind of an infinite player creator class for single player.

# MONGODB (*MongoWriter.java*)

In this part I'm going to explain what our Mongodb part does, in such a fashion that anybody with an intention of connecting a remote Mongodb server, updating, inserting and deleting data can benefit. I am also going to talk about how someone can do the things above periodically.

Note: Language used in the next parts is Java since it was the language we used for the project. With that being said, I don't think this is going to be an issue for the general audience as the Drivers for MongoDb are fairly similar between languages.

**Connecting Your Application With Remote MongoDb Server**

You have a few options here. You could run an EC2 instance in AWS and then ssh into it. Then from the command line you could install MongoDB and then try to connect it from your application. What I did was easier: run a cluster in MongoDb Atlas.

It comes with a cluster of MongoDb already installed. But you still need to do few things:

On the security tab, you need to grant access to IP’s for connection.I entered “0.0.0.0/0” as this means All IP’s can connect to your database. But beware that this is not good practice. I only did this since this DB will not have any sensitive data and will be only used for practice.

You’ll also have to create a database user and grant it access to the databases you want. You can customize the user and its privileges in the “Database Access” tab.

After these configurations you are ready to connect your application to your server.

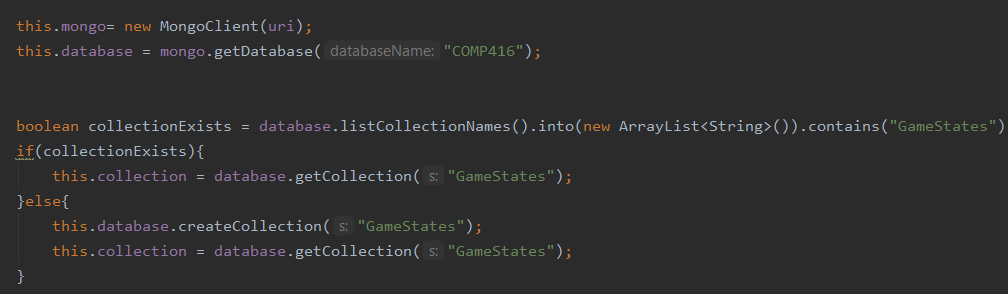


With the previously created database user you can get a link similar to above. After you create a cluster, there will be a “connect” tab where you can find a code snippet with your language of choice. In this tab you can find your URI. I suggest that you store your username and password in a config file or as environmental variables rather than hard coding them.

With the URI of your cluster combined with your credentials you can create a new instance of MongoClient and get your desired database like the code example below.

I also like to reach a collection after making sure it exists in the first place. You can check that by iterating over the list of collection names.

Now you are all set and ready to do some db operations.



**Inserting and Deleting Documents from MongoDB**

MongoDB makes it ultra simple to insert or delete a Document. You can filter all the collection items with Filter.eq which creates a filter that matches all documents where the field is equal to provided value.

After getting the desired Documents you just iterate over them and delete them like below:

(Note that since \_id is unique no iteration is required.)

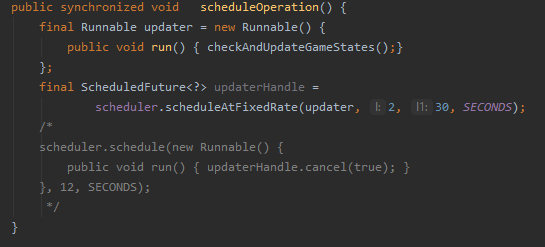
****

Inserting is fairly straightforward as well. Instead of deleting you just call “insertOne”.

**Doing Database Operations Periodically**

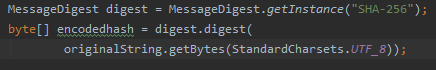
Java has a nice interface called ScheduledExecutorService which enables us to run our methods periodically. What we do here is create a Runnable type object that has the run method which includes what we want to do periodically.

After that we give this Runnable object to scheduler.scheduleAtFixedRate method which returns us a scheduledFuture object. scheduleAtFixedRate takes in a task you want to do, delay and rate. (The same method was also used for deleting files and regularly asking files from the master by follower).



**Detecting Changes in A File and updating The Changed Ones**

We thought of some ways about detecting changes like checking the last modified date, giving an integer to indicate the file is changed, hashcode of the file etc. But in the end what we went with was the hashing. We used the “SHA-256” algorithm that generates almost unique hashes of the SHA family. Java provides access to this algorithm though MEssageDigest class.



All we had to do is create an instance of MessageDigest class and digest it with our string.

Moreover we had to convert this byte array into a hexadecimal string to get the hashed value.

In our case the string used was the string parsed version of our game state file which was in JSON format. We used JSON.parser which is included in the “Simple JSON” library and fed it with our filereader. This resulted in a JSONObject and we simply called its toString method to get a string with the contents of our game state.

**File Synchronization with MongoDB**

We now combine all the things we have covered so far. The goal is to periodically check the files in a directory and synchronize it with our remote database.

In our case we have a MongoWriter object which connects to a remote database when its initialized. During the initialization part we also create a Hashmap of file name, hash pairs.

We insert all the files in the database and keep the ids in a separate id table.

When we call the update method we create a new Hashmap of the current files in the directory with file name, hash pairs. Comparing the entries of the updated map and the previously created map will be used to detect changes.

If a file is in the old map but not in the new map, this means that the file is not in the directory any more. In this case we don't have to store it in the db any longer. We delete it from db with the id thanks to our id table.

If a file is in the new map but not in the old map, this means that this is a new game file.We insert it in db. We also add the newly added object into the id table as well

If a file is in both maps but their hashes are not equal, this means that the contents of that file is modified. In this case, the first thing we do is delete that file based on its id with the help of the id table. Then we insert the file into db as mentioned above. Afterwards we update the id table with the id of the newly inserted document.

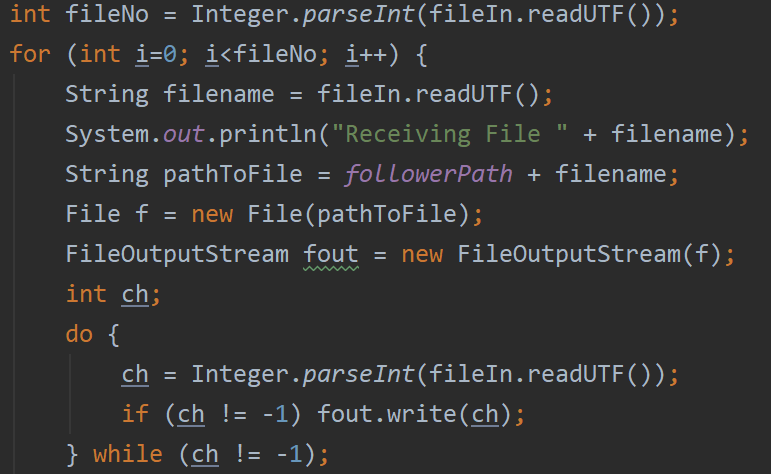
Finally we assign our updated map to be our old map so that in the next call we can compare the changes again.

Our last step is to call a scheduler with our method so that it happens at a fixed rate, say 30 seconds.

# Follower Side

The follower works on two ports, namely the data\_port and the command\_port. Since there can be multiple followers, the directory in which the follower saves its data depends on its follower number. Like the follower, it has a logger that logs both to the console and a unique file based on its follower number. The code for the FTP connection created between the follower and the master is based on the code found on <http://publicvoidlife.com/2014/12/18/file-transfer-protocol-ftp-implementation-java/>. As explained above, when a follower connects to the master, a new followerhandler thread is assigned to communicate with the follower. The follower reads the config file and gets the necessary information (IP, port numbers), and then establishes a connection with the master. It receives its unique followerNumber from master, and creates a directory based on this number. And then it schedules an operation to ask for updated files from the master, and receive if there are any updated files. For scheduling, refer to the scheduling part of the report.

The receivefile method takes in as arguments the relevant datastreams and a boolean, reRecieve. This boolean value is used to signify whether the previous set of changed files were corrupted. It is set to true if that is the case. In this case, the follower doesn’t send the send request as a string again, because the followerhandler also knows that some of the files it sent need to be resent. It receives a number of files that is going to be sent, and for each file, gets the filename, creates a file in the correct path and starts getting the file data from the handler.



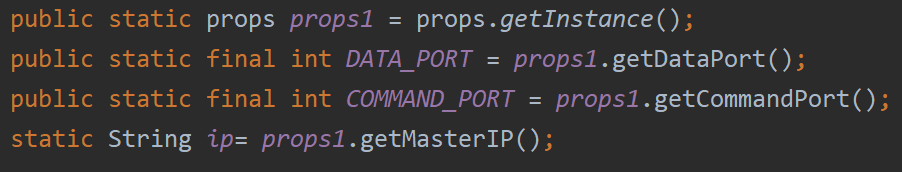
After the file is sent, it gets the hash value of the file and computes the received file’s hash and compares them. The computing of the hash values is explained in another section. If they do not match, the file’s name is added to the If files were corrupted (hashes aren't the same), those files are stored in an arrayList and their names are transmitted to the followerHandler (for resending), after sending the RETRANSMIT signal. And then recieveFile method calls itself, reRecieve value is passed as true. Otherwise, everything is good and CONSISTENCY\_CHECK\_PASSED signal is sent to the handler.

**Config File**

The following link was useful: <https://mkyong.com/java/java-properties-file-examples/>. For logging purposes, we created a class props. It reads from a file called config.properties. The master’s IP, dataPort, commandPort and mongoUser and mongoPass is in config.properties file. The class is a singleton, and all classes access the same instance of the props. It uses java.util.properties and its getProperty method to access the values in the config file. An example is given below.



Below is a code snippet from the follower class. It illustrates how the props class is used to read info from the config file.



**Task Division**

We used the proposed task distribution.

Deniz Karadayı: Master-client interaction and the game logic (TCP communication).

Özgür Budak: Master and MongoDB API interaction (HTTP communication). File writing/deleting and Hashing

Bulut Bulgu: Master-follower interaction (FTP communication).