COMP416 Project#1 Report

Aslı Ağılönü
Doruk Taneli
Onur İskenderoğlu

Introduction

In the project we tried to simulate a basic TCP connection through a network game called War. The War game is a card game where players are dealt the half of a deck and win if the card they draw has a greater value than their opponent. The network model primarily consists of a master server that accepts clients that wish to play the game, the clients that connect to the master to receive their decks and proceed with the game and a follower that backups the game states the master documents. The backup is also maintained through a database connection with MongoDB in which the master updates the game state on the database in case of a failure. This ensures that in the case of a connection loss, the master can retrieve the state of the game and restart without destroying the current game data.

Transport Control Protocol (TCP) ensures the reliable data transfer through establishing a connection between two end points and maintaining that link until the communication is over. The detailed usage of TCP sockets will be discussed in the following parts of this report. In War game, orderly and reliable data transfer is crucial since the message sent by the clients need to be received and interpreted by the master in a proper way to be able to determine the outcome.

Master-Client Connection

Master-Client connection makes up the backbone of the game as it is the part where the game logic is executed. Initially we ask the user if they want to be a master or a follower. If they chose to be a master, a master server is created and then it starts to listen its socket for an incoming connection. In order to support multiple clients to connect to our game, the master server starts a new thread for each incoming client connection. When a client connects, the master deals half of the deck to the client. The code in Figure 1 shows how it is done. The class ServerThread is an instance of Thread and the thread mechanics are implemented in there. Each

thread is responsible of reading from and writing to a client socket they are responsible of. Each thread have their own input and output streams and work independently of the other ones. The

```
try
{
    s = serverSocket.accept();
    if(s.isConnected())
    {
        clientCounter++;
        Deck halfDeck = new Deck();
        if(this.deck.deckSize() > 26){
            ArrayList<Integer> temp = new ArrayList<Integer>(deck.deck.subList(0, 25));
            ArrayList<Integer> temp2 = new ArrayList<Integer>(deck.deck.subList(26, 51));
            System.out.println(deck.deck.subList(0, 25).toString());
            halfDeck.deck = temp;
            deck.deck = temp2;
        }
        if(this.deck.deckSize() <= 26)
        {
            halfDeck.deck = deck.deck;
        }

        System.out.println("A connection was established with a client on the address of " + s.getRemoteSocketAc
        ServerThread st = new ServerThread(s, clientCounter, halfDeck);
        st.start();
```

Figure 1

main difficulty we experienced and couldn't manage to solve at the time of this submission is to ensure the communication between threads. Mainly, they do not need to communicate with each other however the master server should be able to store the data incoming from the different threads (for the duration of the game) and make computations over them. The value of the card for each client is stored in their own threads and currently we couldn't find a way to distinguish between threads and retrieve information. If it were to work as intended, the master will receive the cards each client has drawn, compare them and return the result to each client saying either they won or lost. That part of the project is incomplete and we will attempt to correct it until the demo.

Master and client communicate through sending string messages through I/O streams. We've created a Deck class to implement the game structure. The Deck holds an ArrayList called deck that stores the card values. When a client is connected, the server shuffles the Deck instance

```
            messageRecieved = dataIn.readLine();
            if(messageRecieved.length() > 1)
            {
                ArrayList<Integer> cardList = new ArrayList<Integer>();
                String[] parser = messageRecieved.split(",");
                for(int i = 0; i<parser.length; i++){
if( line.
{
    Strin
    Syste
    for(i
    {
        d
    }
                    cardList.add(Integer.parseInt(parser[i]));
                }

                myDeck.deck = cardList;
                System.out.println("Cards recieved");
            }
    dataOut.println(deck);;
    dataOut.flush();
    System.out.println("Client " + s.getRemoteSocketAddress() + " got their deck");
}
```

it had created and sends the half of it to the client. In order to achieve that, first the contents of the ArrayList is transformed into a String where each element is separated by a c Figure 3 client side receives the message, it parses the string and adds the contents to an Array List and creates its own Deck instance. Figure 2 shows the server side and Figure 3 shows the clier Figure 2 s implementation. Since all the messages except where the deck is sent has the String length of 1 (being a single number), it is easy to differentiate.

Master-Follower Connection

Master-follower connection is made up of two connections: command connection and file transfer connection. Command connection is done with TCP, and file transfer connection is done with FTP. Follower keeps the game state file, and requests it again every time a change happens in the game state file, so it can stay in sync with master. While sending the game state, master also sends the hashed value of the file's name, so that the follower can check whether the file that came in was correct and intact.

While the paragraph above explains what should happen in the perfectly completed project, our FTP only tries to send a file from master to the follower. Unfortunately, my IDE has given unexpected exceptions and didn't build dependencies probably, and it was too late to accomplish more when I found this out.

In FTP_Server, the filename is taken, and the contents of that file are sent line by line with an ObjectOutputStream across the connection. After the file is completely sent, the buffers and streams are closed to prevent any leaks.

MongoDB Connection

How to connect a java application to MongoDB:

1. Install MongoDB, following the guide in this link: https://docs.mongodb.com/manual/installation/

2. Convert your project into a Maven Project:
   Right click on the project, Configure -> Convert to Maven Project
   New files will appear on your project.

3. Add MongoDB and Simple Login Facade for Java (SLF4J) dependencies to the pom.xml file.

```
<dependencies>
    <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>4.0.0-beta1</version>
    </dependency>
    <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-legacy</artifactId>
    <version>4.0.0-beta1</version>
    </dependency>
    <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.5</version>
    </dependency>
</dependencies>
```

4. I recommend creating a separate class for interaction with MongoDB. Other classes do not have to know the details of the interaction. Feel free to modify this for your needs, I will now explain how I implemented this class taking guidance from https://www.tutorialspoint.com/mongodb/mongodb_java.htm.

5. Constructor: The constructor handles the necessary connections to use the database.

* Creates a MongoClient in localhost with port number 27017.
* Sets credentials to 'myDb' database as username: 'sampleuser' and password: 'password'
* Creates 'myDb' database and creates 'myCollection' collection inside it.

6. Insert: creates a document from the provided game information from the parameters and inserts it into the database

```
* Insert the game info to the database. Only use this function when you first start the game.
* See syncDB function to synchronize the database contents with current game info as the game
      continues.
*
* @param gameID id of the game, typically 1, increase as there are more games being played
      concurrently
* @param P1 name of player1 as string
* @param P2 name of player2 as string
* @param round which round of the game
* @param cardsRemaining remaining number of cards in each player's hands
* @param scoreP1 score of player1 as int
* @param scoreP2 score of player2 as int
```

7. Sync:

```
* Call this function every 30 seconds or when the game state changes.
* checks if the database needs updating
* if needed, updates database according to given parameters
* and returns string with the details of synchronization.
*
* @param gameID id of the game
* @param round which round of the game
* @param cardsRemaining remaining number of cards in each player's hands
* @param scoreP1 score of player1 as int
* @param scoreP2 score of player2 as int
* @return string with details of synchronization
```

8. Get game information from database:

```
* Returns information of the game as Document.
* Use .get("field") to get individual fields.
*
* @param gameID id of the game
* @return game info as Document
```

Task Distribution

We used the suggested task distribution:

Aslı Ağılönü:	Master-client interaction and the game logic (TCP communication).
Doruk Taneli:	Master and MongoDB API interaction (HTTP communication).
Onur İskenderoglu:	Master-follower interaction (FTP communication).

Conclusion

The project was a long and hard one, however it has taught us many different skills, like TCP/FTP connections, database and backup structures, etc. These skills can be used in almost every project that is relevant to network-related systems.

References

Codes from PS1 and PS2 are used for creating TCP connections

MongoDB.java class was written with the help of the Java MongoDB tutorial link provided in the recommended resources. https://www.tutorialspoint.com/mongodb/mongodb_java.htm

http://www.shouttoworld.com/file-transfer-protocol-ftp-implementation-java/