COMP 416
Spring 2020

Project 1: WAR Game

Ahmet Uysal

İpek Köprülülü

Furkan Şahbaz

Contents

1 Introduction

A TCP-based version of the proposed WAR game was implemented for this project, in order to obtain a basic knowledge of the application layer of the network protocol stack. WAR is a card game that consists of 52 cards and 2 players, and is based on the following rules:

- 2 players must be present in order to play the game.
- At each turn, players decide to play a card, start a new game, or quit the ongoing game.
- Each time a player decides to play a card, a random card from the player's deck is selected and the cards played by both players are compared in order to get a result.

The game state is regularly backed up with Followers following the FTP protocol by using JSON files, and is also by using the MongoDB API.

2 Implementation Details

2.1 WAR Protocol

WAR Protocol is an application layer protocol that is built on top of TCP. War Protocol uses only a single type of message that contains a type and a payload. The type of the message is simply a byte, whereas the payload is a byte array. Additional message types were defined in our implementation for opereations required for the developed protocols.

2.1.1 Extra Message Types

Message Type	Value
matchmaking	5
correspondent connected	6
file hash	7
receive file with name	8
file transmit validation	9

- matchmaking message is sent from the players, i.e. the Clients by inputting their names to indicate that they're ready to play the game.
- correspondent connected message is sent to indicate the connection of the correspondent, which is typically either a Follower or a Client.
- file hash message is used while sending the hash values of the file for validation.
- receive file with name message is sent by the Server to the Follower, to indicate the name of the file to be sent, for further synchronization.
- file transmit validation message is sent to perform file validation.

2.2 Server Side

The Server side of the project has to main functionalities: Master and Follower.

2.2.1 Master Functionality

When the Server side of the project is run with the "Master" input, it acts as a Master throughout the execution. For every connection, the master creates a server thread, and sends each message delivered by listening to their sockets to the controller thread's message queue for it to send each corresponding message type to the service for handling various types of actions.

The most critical actions of the Master functionality are sending, receiving, and interpreting messages or files that are transmitted and received. For Followers, the Master mainly focuses on transmitting files for back up and their validation information, whereas for clients, the Master carries on the game logic, while handling any connection-based errors that may occur.

The Master functionality of the server also interacts with the MongoDB API regularly as a backup mechanism, in a fashion similar to its interaction with the Follower side.

If the end-users choose to do so, multiple active games can be connected to the Master. Each game's data will be saved in different JSON files with names of player1-player2.json and will also be stored in the Mongo database. In the case of termination of any game, corresponding JSON files and database documents will be deleted as well.

2.2.2 Follower Functionality

When the Server side of the project is run with the "Follower" input, it acts as a Follower that connects to the Server (its Master), and interacts with it to back up and synchronize the game files regularly throughout the execution.

When a Follower connects to a Server, it sends the Server indicating that it is a Follower to avoid further complication in communication.

2.3 Client Side

The Client side of the project connects to the Server side, in order to communicate with it during game execution. When a Client is first connected to a Server, similar to what the Follower does, it sends the Server a message, indicating that it is a player, i.e. a Client.

After two players have been connected to the Server, they receive a prompt asking for their names in order to start the game. After obtaining both players' names, the game is initiated as the Server provides both players with their randomly selected decks of cards.

At each turn of game, the Clients are provided 3 options to continue with:

1. Play Card: If the player chooses to play a card, a randomly selected card is sent to the Server to decide on the outcome of this round. After the outcome is determined, both players are notified of the result of the game. If the decks of the players are empty, this means that the game has ended as well, therefore the game result is also sent to the players.

2. Start a New Game: If the player chooses to start a new game, the initial game will first be terminated for both sides, and a new game will be started for the user that requested a new game. After this process, the opponent will be deemed as the winner of the previous game, whereas the other one will be waiting for a new opponent.

3. Quit Game: If the player chooses to quit the game, the game on both sides will be terminated. However, the opponent will first be notified that it has won the game.

3 Backup Mechanisms

3.1 Interacting with MongoDB

When the Master functionality of the Server component is initially run, first the ControllerThread is initiated, which then interacts with the service layer of the project, WarService; which uses the repository of the project, namely WARRepository is initiated.

To make our implementation more modular and decouple service and repository layer, the WARRepository was defined as an interface which defines basic operations to insert, update, retrieve, and delete games.

Since the project requires database implementation with the MongoDB API following the HTTP Communication protocol, another class was defined, namely MongoDBWARRepository. This class implements the previously defined WARRepository interface in order to perform the operations both required by the interface definition and the game backup mechanism.

While interacting with the warRepository in the WARService class, it is initiated as a MongoDBWARRepository object by taking advantage of polymorphism. The constructor of the MongoDBRepository class obtains the database credentials from the provided configuration file, and connects to the MongoDB database using these credentials, details of which can be found in code snippet xx. An important step in the initialization of the database was configuring the codec registry settings of the database, since it will be required in order to insert the game object directly to the database as a document.

When a game is first initialized, it is inserted into the warRepository object which is now a MongoDBWARRepository object. For insertion, a Document object from the given file should be generated, using the ObjectId of the game and the whole WARGame class data. After generating this document, it is inserted into the collection of the database that the games are kept, i.e.the WARGames collection. While generating the document, the WARGame object is directly inserted in the database, accompanied by its ObjectId. An example of an inserted, recently initialized, WARGame document can be observed in the following figure:

It is clear to see from the above Fig. 1 that the following properties have been stored:

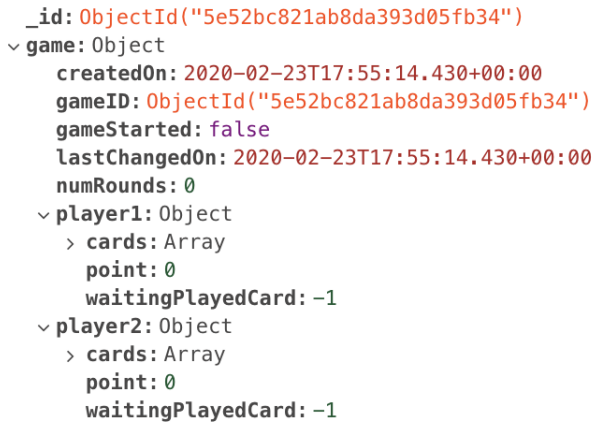

Figure 1: Recently initialized game document in MongoDB

- Both players contained in the game, and therefore their names, remaining cards in the decks, and their scores (since the Player class includes these values).
- The number of rounds played, and corresponding time stamps regarding the updates made on the game document within the database.

The logic that updates the game in the database by regularly checking whether any change has been made in every 30 seconds is implemented in the controller thread. Whenever a change is made to the game state throughout the project, the lastChangedOn field of the game is updated with the time of the corresponding change. This updated value is then compared to the lastUpdatedOn value contained in the controller thread, and if the lastChangedOn date occurs after the lastUpdatedOn date, the game data is updated in the database, by basically querying a database object with the same ObjectId that is provided by the game, and overwriting the document data corresponding to the same ObjectId.

If a retrieval of the previous game data is required at any point of the game process is required, it can be accomplished by the retrieveGame method, which obtains the document corresponding to the given ObjectId in the database, and initiates a WARGame object that loads the document's data can be used.

After each game termination, the document in the database that corresponds to the terminated game is deleted, by simply deleting the document corresponding to the ObjectId of the provided game.

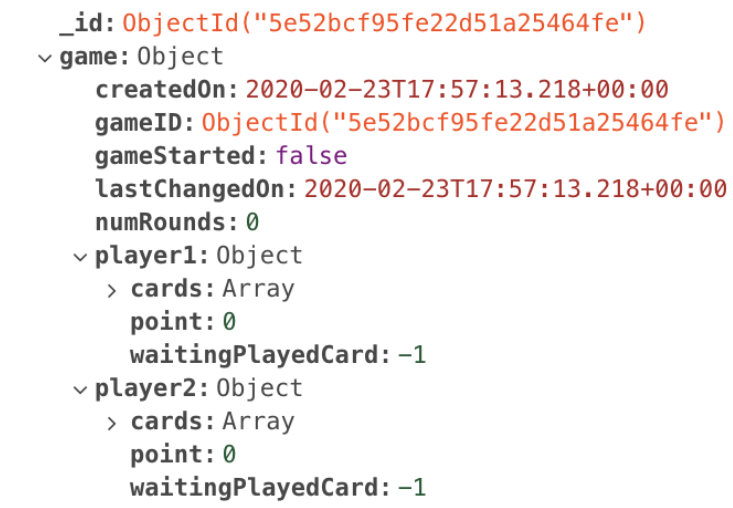

Figure 2: Recently updated game document in MongoDB

3.2 Synchronisation with Followers

When a Server runs as a Follower, it checks the folders with the format of "Follower-ID" and determines the following possible ID to create its own folder. Since every correspondent notifies Master about its type, master can keep track of all Followers and transmits the files regularly. Master keeps the last updated times of the JSON files and the Followers and ensures the synchronisation of the Followers by comparing the update times. This optimises the amount of network resources used for synchronisation.

When the Master wants to send a JSON file to a follower, it first sends a "receive file with name" message to transfer the name of the file, transmits the file through FTP, and sends the MD5 Checksum value of the file for validation. Follower reads the messages and the file correspondingly, creates or writes into the file with the name Master has sent, gets its checksum and compares with the checksum that Master has sent. It sends a message back to Master to inform if the sent file is valid and if not, Master keeps sending until Follower validates the file.

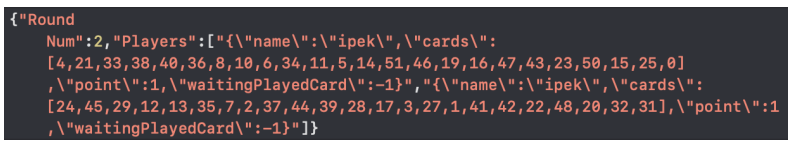

Figure 3: Recently updated game JSON document

4 Task Distribution

We have decided to follow the proposed task distribution while implementing the project:

1. Ahmet Uysal: Master-client interaction and the game logic (TCP communication).
2. Furkan Sahbaz: Master and MongoDB API interaction (HTTP communication).
3. Ipek Koprululu: Master-follower interaction (FTP communication).