COMP/ELEC 416 : Computer Networks

Project #1

Due: February 21, 2020, 11.59pm (Late submissions will not be accepted.)

This is a group-oriented (3-student) project. We suggest a fair task distribution among group members at the end of this project description.

Protocol Design and Development of a Networked Card Game

Introduction and Motivation:

This project work is about the **application layer** of the network protocol stack. It involves application layer software development, client/server protocol, application layer protocol principles, socket programming, and multithreading. Through this project, you are not only going to develop a simple TCP protocol based networked card game from scratch but also you are going to interact with the application layer Application Programming Interface (API) of MongoDB. Also, you will be using the FTP protocol for the file transfers between the master and the follower as described later.

You will be learning and practicing with multiple concepts in this project. One is how to create the TCP server and provide any service (a game in this scenario) to the clients. Another is FTP usage for file transfer. This will be used for the synchronization between the master and the follower. The master/follower will introduce you to the concept of replication and synchronization. Over the Internet, usually, multiple replicas of a main server are kept for better performance and availability.

Project Overview:

In this project, you are asked to develop a networked game named WAR based on TCP master/follower model. The master/follower model is one sort of the server/client model that was discussed in the lecture/problem session. Master server provides TCP connection to interact with the client and FTP connection for the follower. Fig.1 shows connections for a sample master and follower. As shown in the figure, the master server also takes the responsibility of interacting with MongoDB database using the API.

WAR: A Card game

The original card game War, however, is much simpler than that game (although probably not more popular). For this project, we will be programming a cross-platform implementation of the original “war” card game. If you implement the protocol correctly, your code should be able to communicate with other groups’ code regardless of the choice of language.

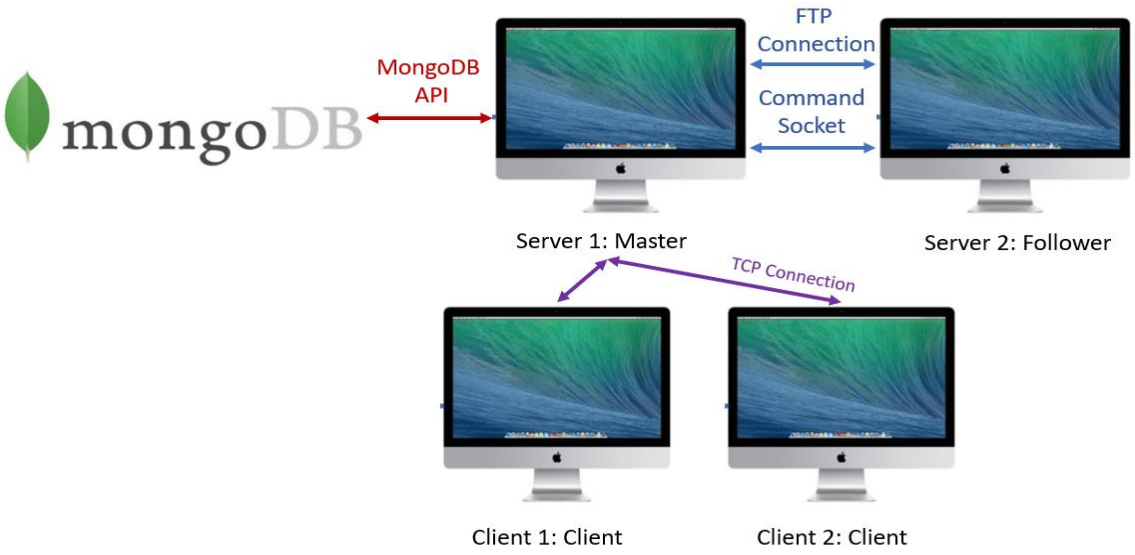

Fig 1. The WAR card game connections

WAR: The simplified rules and the message format

The simplified rules for our version of War are as follows: the dealer (master server) deals half of the deck, at random, to the two players (clients). Each player “turns over” (sends) one of their cards to the server, and the server responds to each player “win” “lose” or “draw.” Unlike normal War (as this project is about computer network programming not video game programming), in the event of a tie neither player receives a “point” and play simply moves on to the next round. After all of the cards have been used, play stops and each client knows (based on the number of points they received) whether they won or they lost. Once each player has received the results of 26 rounds, they disconnect.

All WAR game application layer messages follow the WAR message format.

Message type	**value**
want game	0
game start	1
play card	2
play result	3
game result	4

For “want game”, “play card”, “play result”, and “game result” messages, the payload is one byte long. For the “game start” message (where the payload is a set of 26 cards), the payload is 26 bytes long representing 26 cards. The byte representation of cards is a mapping between each of the 52 cards in a standard deck to the integers [0..51]. Mapping cards follows the suit order Clubs, Diamonds, Hearts, Spades, and within each suit, the rank order by value (i.e. 2, 3, 4, … , 10, Jack, Queen, King, Ace). Thus, 0, 1, and 2 map onto 2 of Clubs, 3 of Clubs, 4 of Clubs; 11, 12, 13, and 14 map onto the King of Clubs, the Ace of Clubs, the 2 of Diamonds, and the 3 of Diamonds; and finally 49, 50, and 51 map onto the Queen, King, and Ace of Spades. You need to write a custom comparison function which maps two different card values onto win, lose, or draw. The mapping examples are shown below.

Integer Value	Mapping to the card
0	2 of Clubs
13	2 of Diamonds
26	2 of Hearts
39	2 of Spades
The rest of the cards will follow the series.	

When sending a “game start” message, the server sends half of the deck, at random, to each player by sending 26 bytes with the values [0..51] to one player and the remaining 26 to the other player.

When sending a “play card” message, the client sends one byte which represents one of their cards in a random manner. The following results value explain the payload of “play result” or “game result”, meaning instead of sending “win” you will send “0” to the winning player. “Play result” is the result of the round played and the “game result” is the overall result of the game (i.e. who won the most rounds).

Result	Payload
win	0
draw	1
lose	2

WAR: The network protocol

With the simplified rules, the WAR protocol is as follows. A war server listens for new TCP connections on a given port. It waits until two clients have connected to that port. Once both have connected, the clients send a message containing the "want game" command. If both clients do this correctly, the server responds with a message containing the "game start" command, with the list of cards dealt to that client.

After each client has received their half of the deck, the clients send messages including the "play card" message, and set the payload to their chosen card. After the server has received a "play card" message from both clients, it will respond with a "play result" message to both clients, telling each one whether they won or lost. This process continues until each player has sent all of their cards. After both clients send their final cards, the server decides the winner of the game and sends the "game result" message to each of the clients, declaring the winner of the game. The server disconnects after sending the "game result" messages, and the clients disconnect after receiving them.

Implementation Details

This project has the following main components, as described next: 1) Master side of the game, 2) Follower side of the game, 3) Clients for playing the game, and 4) Interactions with MongoDB.

Master side requirements:

Master server should:

- have TCP-based implementation of the game mentioned above.
- wait for two clients to connect before it starts the game between them.
- keep the state of the game in a file named *"Name of player1- Name of Player 2.json".* The master will only keep the state of the game until it is over. Once the game is over, the state is removed. The game state should be stored regularly each 30 seconds.
- establish an API Client connection to MongoDB. All the game state will also be kept on the MongoDB for backup. The game state should contain the following information.
    - Name of the players.
    - Number of rounds played.
    - Remaining cards of each player.
    - Score of each player: Count of the rounds won by each player.
- implement FTP protocol for the file transfer between the follower and the itself.
- be **multi-threaded** and should support both multiple followers and multiple clients. For multiple clients, there will be a seperate game for each pair of clients.
- interact with MongoDB in a timely fashion (e.g., every 30 seconds), which is called the synchronization time window, when it is executed, detect any changes in the game state and if any update MongoDB accordingly. Each 30 seconds, the following protocol should be followed:

    1. Check if the synchronization is needed. This message should be logged on the server:

    "Current time: <current exact time>, no update is needed. Already synced!"

    or

    "Current time: the following files are going to be synchronized":

2. If yes, then synchronize and display following message on the server:

   "Synchronization done with MongoDB"

Note: To keep the game state synchronized with MongoDB, you need to come up with a way to detect changes in the game state. Only when changes are detected, the MongoDB database should be updated with the new game state file information.

Follower side requirements:

In this project, you will learn and implement different ways to store your application state persistently and reliably against machine failures. One way is writing the game state to a file stored locally. A second is storing the state in a (possibly remote) database. A third is to have an active server (the master) and one or more idle ones (the followers) that keep the same state. In case of master failure, a follower would take over to continue serving the game with minimal service disruption. For this project, you are only required to support having one master with one or more followers, and to keep the follower state(s) up-to-date by doing regular synchronization. For simplicity, we do not ask you to handle any failure scenarios.

In this part, you will implement a follower server. The follower communicates with the master only and has no communication with MongoDB. You may achieve the synchronization between master and follower(s) by keeping track of the game state files, or by coming up with your own method of synchronization. For example, you may exchange the hash of the files to track the changes. **Whatever method you use, only the updated and newly-added files should be exchanged**. Sending unmodified files over and over again is a waste of network resources.

There should be two connections between the master and each follower for exchanging data and commands. The data exchanges between master and follower is for exchanging game state files and should be done over the FTP connection as shown in Figure 1. The second connection, on the other hand, is for the commands. For example, the follower can ask for state changes through the command connection, and the master sends the changes through the data connection. The command connection should also be used for the verification of the files transfered. Once all the file is sent, the master should take the hashed value of the file and send it on the command socket. On receiving the file, the follower should also listen to the command socket and receive the **hashed value of the file**. The follower should check the correctness of the file transmission by taking the hash value of each received file and compare it with the corresponding hashed value **received from the master**. If there is a mismatch in the hash value of a file, the entire file should be requested again by the follower. After transmission of the file and hashed values is complete, the master should listen to the command socket, until it receives one of the following messages from the follower:

- "CONSISTENCY_CHECK_PASSED": This means the entire file has been transmitted successfully, and both master and follower can close their data socket.
- "RETRANSMIT": This message alarms the master that the data received were inconsistent and corrupted, and the sender should retransmit the entire file again. This procedure is repeated until the receiver sends a "CONSISTENCY_CHECK_PASSED" message.

Sent and received messages should also be logged at both the master and follower sides. These messages are helpful for your project demo. Moreover, there will be multiple followers.

Client side requirements:

You will be implementing the client side to play the game. The basic design is:

- Client will open a TCP connection to the server, and use the TCP protocol to communicate with the server.
- Client will send its name to the server and then receive the 26 cards from the server.
- Upon receiving the cards, there should be a message saying “Cards received” displayed.
- The client should get the following options upon receiving the cards: 1) Play the card, 2) Start a new game, 3) Quit the game. These options are repeated each time the client does a move.
- Upon starting the new game, the previous game state should be removed.
- Upon quitting the game, other users should be declared the winner automatically.

Interaction with MongoDB:

MongoDB will be used to keep the state of the game. Following links can be useful for its use:

Installation: https://docs.mongodb.com/manual/installation/

Java MongoDB tutorial: https://www.tutorialspoint.com/mongodb/mongodb_java.htm

System execution scenario requirements:

In the project demonstration, you will be asked to show an execution scenario with the given requirements.

1. Follower and Master should both be unified in the same project. Upon execution, your implementation should ask the user to determine either master or follower mode.
2. All the parameters (e.g., address of master, MongoDB credentials, etc) that concerns the user should be stored in a configuration file and used at startup.
3. There are three separate computers connected to the same Wi-Fi router (i.e., access point).
4. The system should run on at least three computers. One computer would act as master, one would act as follower and the (multiple) clients will run on the third computer.
5. You may be asked to run several concurrent followers (running on different threads) that querying the server. Note that if you are executing more than one follower on a single computer, your code should automatically put the files for different followers in different folders e.g., if there are two followers running on the same machine, the files of the followers should be in Follower1 and Follower2.
6. Master will handle all the game executions.
7. Once the master starts, it will wait for two clients. Once two clients are connected, the game will begin between them in a separate thread. The clients are connected randomly and the server should keep track of which two clients are playing with each other. Then, the Master will again start waiting for the next two clients.
8. Master will keep the game state of all the running games in separate files.
9. The game state will automatically be removed once the game is over.

Bonus part:

For the bonus, you are asked to implement the server so that the game can be played using the browser installed on your local machine. For the bonus, you have to handle the HTTP headers that will be part of the request sent by the browser. You will also have to implement an interface to be able to play the game over the HTTP connection.

Project deliverables:

You should submit your source code and project report (in a single .rar or .zip file) to the Blackboard.

- Your entire project should be implemented in Java. Implementation in network-based languages (e.g., Ruby) is not allowed. For implementation in other languages you should first consult with TA and get confirmation.
- Source Code: A .zip or .rar file that contains your implementation in a single Eclipse or Intellij IDEA. If you aim to implement your project in any IDE rather than the mentioned one, you should first consult with TA and get confirmation.
- The **report** is an **important part of your project presentation**, which should be submitted as both a .pdf and Word file.**Report acts as a proof of work for you to assert your contributions to this project**. If you need to put the code in your report, segment it as small as possible (i.e., just the parts you need to explain) and clarify each segment before heading to the next segment. For codes, you should be taking screenshot instead of copy/pasting the direct code. Strictly avoid replicating the code as whole in the report, or leaving code unexplained. You are expected to provide detailed explanations of the following clearly in your report:
    - How does your master connect to MongoDB, and update the information? Your answer should be generalized as a step by step guide for anyone who aims to connect a Java application to MongoDB, check for updates, and do uploads and downloads.
    - How does your follower synchronize itself with the master? You should explain your proposed synchronization protocol in a clear and smooth language first, and then the code correspondence of it.
- In your report, you need to dedicate a section describing the task division among the group members. There you are supposed to clearly state the contributions of each group member to the project.

Demonstration:

You are required to demonstrate the execution of your game protocol with the defined requirements. Your demo sessions will be announced by the TAs. Attending the demo session is required for your project to be graded. All group members should be available during the demo session. The timely attendance of all group members to the demo session is considered as a grading criteria.

Important Notes:

- Please read this project document carefully BEFORE starting your design and implementation.
- In case you use some code parts from the PS codes or the Internet, you must provide the references in your report (such as web links) and explicitly define the parts that you used.
- You should not share your code or ideas with other project groups. Please be aware of the KU Statement on Academic Honesty.
- For demonstration, you are supposed to bring your laptops and run your program on your own laptops.
- Your entire code should be commented as well as JavaDoc provided properly. You may use the following reference to get ideas about writing a clear and neat JavaDoc. Quality of your provided comments and JavaDoc is considered as part of your overall grade.
  https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

- Your report should present your design aspects and it acts as a reference for your implementations. Please take the report part seriously and write your report as accurately and complete as possible.

Suggested task distribution:

For a group of 3 students, we suggest the following task distribution:

Student 1: Master-client interaction and the game logic (TCP communication).

Student 2: Master and MongoDB API interaction (HTTP communication).

Student 3: Master-follower interaction (FTP communication).

Good Luck!