Assignment 1:
A Client-Server Phrase Guessing Game

Due Date: Tuesday Feb 7, before midnight
Demo: (details for registration for demo will be announced).
Weight: 10%

Description

For this assignment, you must develop a client-server distributed application in Java for a phrase guessing game. The client connects to the server and specifies the game level they want to play. The server then chooses a number of words from a dictionary based on the game level, and asks the client to guess the phrase. The client (the player) tries to guess the words chosen by the server by suggesting letters (one letter at a time) or the whole phrase. If the client suggests a letter that occurs on the phrase, the server places the letter in all its positions; otherwise the counter of allowed failed attempts is decremented. At any time the client is allowed to guess the whole phrase. A failed attempt occurs either when a suggested letter does not appear in the phrase, or when the suggested whole phrase does not match.

The client wins when the client completes the phrase, or guesses the whole phrase correctly. The server wins over when the counter of allowed failed attempts reaches zero. The server should compute the total score of games using a score counter: if the client wins the score counter is incremented, if the client loses the score counter is decremented.

The server should keep track of client game history so returning clients are able to see their previous score, and new rounds of games update client score.

The server upon initialization builds a repository of words based on a standard Unix words file. The server must allow the client to add words to the repository, or remove a words from the repository, or check if the repository already contains a given word.

Implementation Details

Client and Server Programs Create a server that binds to a local port, and a client program that connects to the server with given host and port as command line arguments. (refer to your tutorial task 2 for examples, and lecture slides 04_CommunicationParadigms_P1. . . .pdf). The client must provide a simple command line interface for the player (more details below).

Communication Protocol The client and the server communicate by sending messages over a TCP connection. First think carefully how to design the communication protocol for your client-server system. Clearly write down the content, format (syntax), and order of the messages that are sent from the client to the server, and from the server to the client. Clearly indicate the actions that follow from the receipt of these messages on each side. The following is a starting point for the game playing actions. Extend and improve it as you see fit.

- *Start of connection*
  - **Request:** The client should send a registration or sign-in request that includes the client name (how can you make this more secure?).
  - **Reply:** The server should send a "welcome" or "welcome" back message to the client, followed by the list of available commands. This list must include (but is not limited to) commands to start and play a game, check client score, and commands to add, remove and check whether a word is in the repository.
- *Start of game*
  - **Request:** If the client (player) wants to start a new game it sends a special "start i f" - message to the server, where the level i indicates the desired number of words in the phrase, and f is the number of failed attempts factor.
  - **Reply:** After the server receives the start of game message from the client, it asks for i random words from the word repository service, and concatenates them to create the phrase for a game round (words are delineated with spaces in a phrase). The server sets the counter for the number of failed attempts to $i * f$. The server then replies with a message that includes a row of dashes (a current view of the words), giving the number of letters in each word and the spaces between the words.

    For example, if the client has started the game with "start 4 4" (i.e., $i = 4$ and $f = 4$), the server may choose the following phrase "distributed systems is fun" and sent the failed attempt counter to 16. The server then sends the following row of dashes and counter value to the client: "`----------- ------- --- ---`C16".
- *Game play: client to server*
  - **Request:** Upon receiving the phrase and counter, if the client wants to suggest a letter or the whole phrase, it sends the letter or the guessed phrase to the server. At any point in the game, the client can check its score by sending the '$' character, choose to start a new game by sending the '!' character, and end the game by sending the '#' character. At any time, the client can check if a word exists in the repository, by sending the '?word' message.

- **Reply:** The server replies depending on the case:
  * If the client is checking for its score, send the total score.
  * If the client is checking whether a word exists in the repository, the server must reply with an appropriate message to indicate if it could find the word in the repository or not.
  * If the client guesses the whole phrase correctly, the server sends a congratulatory message with the total score;
  * If the letter guessed by the client completes the phrase the server sends a congratulatory message with the total score;
  * If the letter guessed by the client occurs in the phrase and the whole phrase is not yet completed, the server sends to the client the current view of the phrase with the letter placed in all its correct positions and the current value of the failed attempts counter.
    For the example phrase given above, if the client starts by suggesting ‘g’, then the current view is still all dashes and the counter is decremented, i.e., “`----------- ------- --- ---`C15”. If the client then suggest ‘s’, then the current view of the phrase and the failed counter become “`--s-------- s-s---s --- ---`C15”. Next, if the client suggests ‘m’, the current view of the phrase and the counter become “`--s-------- s-s--ms --- ---`C15”;
  * If the letter guessed by the client does not occur in the phrase or the client guesses the whole phrase incorrectly, the server decrements the failed attempt counter and, depending on the counter’s value, sends either the current view of the phrase together with the value of the failed attempt counter (if the counter > 0), or a “Lost this round!” message together with the total score (if the counter $== 0$).

User Interface for the Client A simple command line client user interface should allow the user to start a game, input their choice of a letter or the whole phrase, or add, remove and check whether a word is in the repository.

During the game play, the following information should be displayed to the player:

- a current view of the phrase;
- the current value of the failed attempts counter;
- and the current value of the total score.

Microservice for Word Repository

You should develop the word repository as a microservice that offers its services via UDP-based communication. Recall that UDP is connectionless and unreliable, but simple and fast. The word repository server must bind to a UDP port and listen for incoming requests. You should design the communication protocol for the following services provided by the word repository:

1. add a word
2. remove a word
3. check if a word exists
4. get a random word

You can further expand this list by providing a service to return a random word of a certain length (optional).

To initialize the word repository, you can use the "words.txt" file which contains 25143 words. (available on blackboard under assignment resources, or on Linux machines from /usr/dict/words). For the purpose of this assignment, any random collection of words is considered a phrase. But if you like, you can challenge yourself to find other sources to create meaningful English phrases.

Server Handling Multiple Players Simultaneously After you have made sure your sever is able to handle a single client, extend it so that it can handle multiple players simultaneously.

Microservice for User Accounts You should implement a microservice for keeping track of track of the user accounts and game history. You can decide whether this service is accessed via UDP or TCP.

Bonus: Indirection Server To further challenge yourself, turn the game-play logic of the server program into another microservice, and run it as its own micro-server (you can decide whether it should provide the service over UDP or TCP).

Create an indirection server that coordinates communication between the client and the game-play and word repository microservices. The indirection server must use TCP to communicate with the clients. When a connection is established with a client, the indirection server receives a command from the client and then communicates with the appropriate microservice to perform the user's request. The indirection server then returns the reply from the microservice back to the client via the TCP connection. The client can close the TCP connection to the indirection server when it is done.

Documentation and Coding Standards

There is no official course standard for documentation and coding style. However, your code must be clearly documented, and written following a consistent formatting for indentation and naming styles. Your code must be self-documenting, well formatted, logically structured and easy to modify. The exact method by which each procedure/method works must be concisely but clearly and fully explained. Exactly how to do this is left to your discretion, but seek guidance from your instructor if you are unsure. Poorly formatted or documented code will loose mark on this component.

What to Hand In

Hand in a single packaged zip file that includes the following:

1. The communication protocol design document for your distributed application.
2. Your program source code and a readme.txt file with instructions for running the program. You must include the source code for all the classes you have developed, including the client, the server, the word repository, the user accounts, and any other helper classes you have developed.
3. This assignment may be completed in groups. Each group must submit a statement of contribution that clearly states the contribution from each team member. Submit your code package and your statement of contribution to the D2L dropbox for assignment 1.

Outcomes

Once you have completed this assignment you will have experience with following:

- You can develop a distributed (client/server) application, i.e. define tasks, decompose the server side tasks into microservices, assign tasks to processes and develop an application-specific communication protocol for process interaction using TCP and UDP sockets;
- You can use Java classes representing TCP and UDP sockets (the java.net package) and IO stream classes (the java.io package) to program and to use message passing in a distributed application;
- You know how to use Socket and ServerSocket classes of the java.net package for communication between a server and clients;
- You know how to use byte streams of the java.io package for sending/receiving protocol messages over a TCP connection such as InputStream and OutputStream, BufferedInputStream and BufferedOutputStream. And you know how to use streams, stream and string tokenizers of java.io and java.util package for programming a character-based protocol, such as PrintWriter, BufferedReader, StreamTokenizer and StringTokenizer; and

- You can program, create and use concurrent threads in processes (clients, server) of a distributed application in order to improve scalability and performance (e.g. response time), to hide communication latency.

Rubric (100 marks)

1. Documentation 10
2. Clear communication protocol design 10
3. Client implementation 30
4. Server implementation 50
   (a) 20 implementation of interaction with a single client and game play logic
   (b) 10 sever can handle multiple clients (e.g., via multi-threading, keeping track of the state of the game for multiple clients)
   (c) 10 implementation of user accounts as a microservice
   (d) 10 implementation of the word repository as a microservice
   (e) 10 (bonus) implementation of server with indirection