Assignment 4
Threads and Serialization

Version: March 2, 2023

Prerequisites

1. The assigned readings in module 4 on Canvas
2. Lecture videos from Canvas (or in person)
3. Running and understanding the examples listed on the Canvas page
4. Setup of a second device (second computer, AWS EC2, Raspberry PI) – see Canvas for details (should already be done)

Learning outcomes of this assignment are:

1. Apply sockets in an efficient way
2. Understand how to use threads
3. Understand how to work with shared state when using threads
4. Use different protocols to serialize data

Preliminary things (10 points)

I strongly advise you to work on Git and GitHub, to version control and also to practice. Submit your code and all documents via your GitHub Assignment 4 directory.
Please watch the videos supplied with this assignment to get some more insight.

What you definitely need:

1. Structure: you will have to create two programs one for each Activity. So your assignment 4 folder should have two subdirectories, you can keep them as they are in the starter code. Each project needs a README.md and a build.gradle file.
2. README.md for each activity
   a) A description of your project and a detailed description of what it does
   b) An explanation of how we can run the program
   c) Explain how to "work" with your program, what inputs does it expect etc.
   d) A short video for each activity (2-4min) showing how you run the program, showing what works and briefly show your code.
   e) Design your calls and user interaction in a way that they are easy. Remember we have many assignments to grade, design it so it is easy for us. There will be some requirements later you should fulfill.
   f) Name the requirements that you think you fulfilled

1 Activity: Threads (30 points)

Background

For this activity you will convert a simple single-threaded server to a multi-threaded server. You will create 2 versions, one that allows threads to grow unbounded, and one that sets the number of allowed clients at a time to a fixed number. This is just a toy example to get started this should not take crazy long.

You are given starting code of a single-threaded server. There is a Client provided, which you should use for your implementation. It also specifies a protocol which you need to implement as is.

Given Code

You are given starter code for this assignment on Canvas:

- Server.java - a main program that accepts incoming client connections
- Performer.java - a program that does the "business logic" of what the client requests
- StringList.java - a simple wrapper class for a list of strings
- Client.java - a client which has the main functions and user interaction part which you should implement on the Server/Performer side
- A build.gradle file which can run the base Client and Server, see Readme

You can have any package structure you want and add new classes etc. Important, you should only have 1 Readme and 1 build.gradle file for all 3 tasks. Each task (server) can be started separately and work only with the features described. You can make any changes in the given files you like, BUT the protocol should work as described in the code and README (and Task 1).

Task 1: Make Performer more interesting (11 points)

Presently, all the Performer does is add strings to an array. Make it more interesting by what the Readme specifies.

Task 2: Make the server multi-threaded (12 points)

You can add new classes here of course. Task 1 should still run as is!

1. Name this server class "ThreadedServer"
2. Allow unbounded incoming connections to the server.
3. No client should block.
4. The shared state of the string list should be properly managed.

Task 3: Make the multi-threaded server bounded (3 points)

You can add new classes here of course. Task 1 and 2 should still run as is. You can make any changes in the Performer, the add is given but it might not use the correct protocol – you can change that.

1. Name this server class "ThreadPoolServer".
2. Only allow a set number of incoming connections at any given time. How many should be specified when calling the program through Gradle.
3. Name this server class "ThreadPoolServer".

Gradle (4 points)

1. In your ONE Gradle file there should be 3 Gradle tasks to run the different servers
2. Gradle should use default values for each task, per default host=localhost, port=8000
3. We should be able to run "gradle runClient" and it should also use the default values.
4. Running your different servers:
   a) One for running Task 1: gradle runTask1
   b) One for running Task 2: gradle runTask2
   c) One for running Task 3: gradle runTask3
5. Provide a detailed Readme.md file in your project, which tells us how to run each task and a description of which parts you accomplished of the requirements.
6. Make sure you include your screencast here as well, one screencast for all 3 parts not longer than 4 minutes.

Put things online

I am not giving points for this but if you like put your server online so others can test and "break" it. You should implement the protocol exactly as given, so in theory all Clients should be able to work with all Servers.
Let others know if you break their server, by doing this part you can get help from others and give help to others.

2 Activity: Threads and Protobuf (60 points)

A simple multi-player game using Protobuf as protocol.

The Game

We want to implement and design a simple game, where users can play a simple Pair Guessing Game with similarities of Memory.

The server is the game host and is the only one knowing the original tiles for the game, a leader board and a log file. The server and clients communicate through a given Protobuf

protocol. See protobuf files and the Readme in the given code.

YOU HAVE TO implement the given protocol as described the given proto file!!

Your code needs to work with our protocol exactly as defined in the Readme, if you change it then our client would not work with your server. Theoretically, everyone's client should work with everyone's server. //
See the video on how the game might look like for more detailed information. //
The points in the constraints section add up to more than 60 points, the extra points will be extra credit. You can basically choose which ones you fulfill. BUT it needs to be a playable game that makes sense and implements the protocol.

Constraints (60 points)

These are estimates, server and client should also not crash, it should be well implemented, readable, use good coding practices. If you do not do the above you might loose points even if the functionality is fulfilled.

1. The project needs to run through Gradle (nothing really to do here, just keep the Gradle file as is)
2. **(6 points) You need to implement the given protocol (Protobuf) see the Protobuf files, the README and the Protobuf example in the examples repo (this is NOT an optional requirement)**. If you do not do this you will loose 15 points (instead of getting the 7 points) since then our client will not run your server for testing and thus basically your interface is wrong and not what the customer asked for.
3. **(4 points) The main menu gives the user 3 options: 1: leaderboard, 2 play game, 3 quit. After a game is done the menu should pop up again. Implement the menu on the Client side (not optional). So the client shows the menu, the Server does not send the menu options to the Client.**
4. (3 points) When the user chooses the option 1, a leader board will be shown (does not have to be sorted or pretty).
5. (2.5 points) The leader board is the same for all users; take care that multiple users do not corrupt it.
6. (2.5 points) The leader board persists even if the server crashes and is re-started.
7. (2 points) User chooses option 2 (play game) and the current game board is shown.
8. (5 points) Multiple users will enter the same game and will thus play the game faster. NOTE: The server will only run a single instance of the game at any given time. If a user requests a new game and a game is in progress, they will join the current game. See video for details.
9. (2 points) Users win after finding all matches, they get 1 point for winning.
10. (2 points) After winning the Client goes back to the main menu.

11. (2 points) Multiple clients can win together and each get a point for winning.

12. (3 points) A tile coordinate is a 2 character string with row first as letter followed by column as number, columns are one indexed (0 is invalid) e.g. tile = "d1", row=d, column=1. You can assume no more than a 4x4 board size, so rows are a, b, c, d, columns 1, 2, 3, 4 – also keep in mind that row * column = even.

13. (3 points) You have a couple boards already given in the resources folder, you are allowed to change the format of these if you do not want the "|" or row column information in them to make turning tiles easier. That is up to you. When displayed on the Client the letters and column numbers should be displayed though!

14. (4 points) Client sends first tile and server evaluates if the request is valid and sends back the board with that tile unturned. Handle errors accordingly, e.g. out of bounds, tile already turned - use the "error" flag in response for errors.

15. (4 points) Client sends second tile (if first tile was success response) and server evaluates if the request is valid and sends back the board with that tile unturned. Handle errors accordingly, e.g. out of bounds, tile already turned - use the "error" flag in response for errors.

16. (2 points) Client presents the information well.

17. (2 points) After both turned tiles are shown user presses any key which will lead to the Client to just show the current board (with the two tiles if not matches turned back again).

18. (3 point) Game quits gracefully when option 3 is chosen.

19. (3 points) Server does not crash when the client just disconnect (without choosing option 3).

20. (4 points) You need to run and keep your server running on your AWS instance (or somewhere where others can reach it and test it) – if you used the protocol correctly everyone should be able to connect with their client. Keep a log of who logs onto your server (this logging is already included). You will need to post your exact Gradle command we can use (including ip and port) on the channel on Slack #servers.

21. (2 points) You test at least 3 other servers with your client. You should comment on their servers on Slack, this is how we will grade these two points. – this can be done up until 2 days after the official due date.

22. (3 points - extra) In my version the board on the client is only updated after they made a "guess", change the implementation so that as soon as the server updates the board the client will get this information and will update the board and inform the user, e.g. client B makes a correct guess, so the board updates and all other clients will get the information about the new board right away.

23. (3 points) If user types in "exit" while in the game, the client will exit gracefully.

24. (2 points) When sending back an error DO NOT just use error codes but a descriptive message that the Client can print. Since others are supposed to be able to use your server they might not know your error codes, so print good messages.

25. Overall your server/client programs do not crash, handle errors well and are well structured. No extra points for this but you might loose up to 5% if this is not the case since your pograms should be robust at this point.

NEEDED: On your server always print the board with all tiles unturned, so we can play faster when grading. Make sure your program is robust with all possible inputs, we should not be able to break it and crash it. We will not be mean when running it but with using it to our best ability and basic "gaming" it should not crash.

Hints:

These are some tips you can take them or leave them.

- Spend some time on the given code, make small changes to it, even if they are stupid. Just to get a feel of Protobuf. Especially with the repeated fields. Get the feel for protobuf before even continuing.
- Start with the base functionality, e.g. just one task one possible answer and so on
- Setup the thread first and not when the rest is done (I did it the other way and it was more painful to change)
- Go little step by little step. I usually add one new request with dummy data, check if I receive it on the server. Then add the real data, check if I receive it. Then I parse it on server, check if that works. Then create response and send it to client, check if I receive it. I usually got a couple lines at a time, especially if it is new to you.

Submission

On Canvas submit your link to your Assignment 4 folder on GitHub and also submit the Assignment 4 folder on Canvas. Remember the Readmes and your screen casts.