Assignment 4
Threads and Serialization

Version: March 8, 2021

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 folder.
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. A 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.

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

Presently, all the Performer does is add strings to an array. Make it more interesting by implementing the following protocol:

1. add <string> - adds a string to the list (presently what it does by default now) and displays the list (strings will be added to the end) – already implemented
2. pop - removes the top element of the list and displays it. If the list is empty return "null"
3. display - displays the current list
4. count - returns the number of elements in your list and displays the number
5. switch <int int> - switch the elements at the given indexes. If one of the indexes is invalid return "null" (list does not change)

Task 2: Make the server multi-threaded

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

You can add new classes here of course. Task 1 and 2 should still run as is.

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

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.

Extra Credit (5 points)

Implement an alternative (additional to task 2) solution to Task 2 using Locks instead of synchronized blocks. Same rules as for the other tasks, should be able to run this through "gradle extraCredit".

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 have to fulfill a simple task and if they do the right thing tiles from an image/text will be turned to reveal the image/text.
The server is the game host and is the only one knowing the original images, 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 in the README!!
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, everyones client should work with everyones 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 but are capped at 70. You can basically choose which ones you fulfill. BUT it needs to be a playable game that makes some sense.

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. **(7 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)**
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).**
4. (3 points) When the user chooses the option 1, a leader board will be shown (does not have to be sorted).
5. (5 points) The leader board is the same for all clients, take care of multiple clients not overwriting it wrong and the leader board persists even if the server crashes (save as file maybe)
6. (4 points) Client chooses option 2 (the game) a new image/text (with xxxx is sent to the client)
7. (8 points) Multiple clients can enter the SAME game and will thus reveal the image faster
8. (5 points) Clients win when finishing an image and get back to main menu, multiple clients can win together (see video)
9. (6 points) Server sends a task and can check if it is done correctly (this needs to be on the server side)

10. (5 points) Client presents the information well and the task are small and fast to answer (one word answers)
11. (3 point) Game quits gracefully when option 3 is chosen
12. (3 points) Server does not crash when the client just disconnect (without choosing option 3)
13. (4 points) Good and fun tasks, that are not the same the whole time and you randomly choose or some other fun way.
14. (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 is already included). I will give it 3 tries over a couple of days, if I can make it through a game and have at least two clients running on it you will get these points. You will need to post your exact gradle command we can use (including ip and port) on the new channel on Slack #servers. 5 days after the due date you need to add your log.txt file to your GitHub repo.
15. (4 points) Do something fun and creative in your game which is not explicit in the requirements but does not go against them. Explain in your Readme.
16. (4 points) Your game has a good setup for how much each task reveals, some kind of dependency on how many games someone won, or how often they logged in. You need to explain this in your Readme and we need to be able to experience it. Do not just turn one tile, an image should be revealed after at least 8 tasks.
17. (3 points) You test at least 3 other servers with your client and should show up on their log file (this is a test and depending how many put their server up that might be hard to do)

NEEDED: On your server always print the answer to the task you are sending to the client, so that we do not have to figure it out while 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.