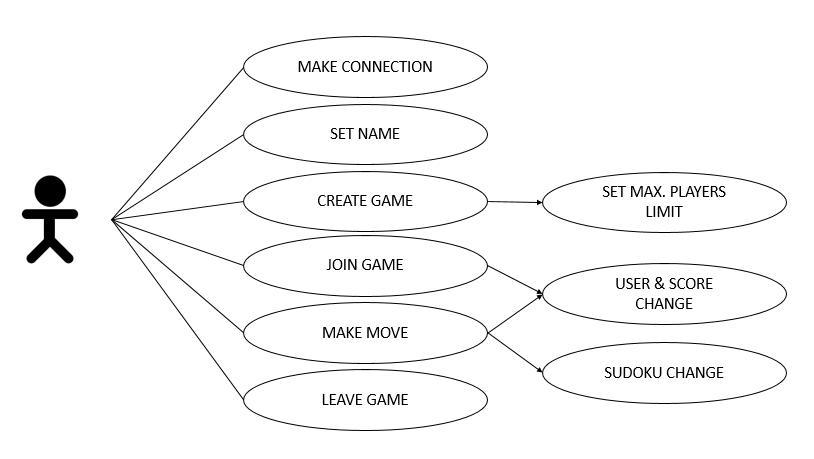
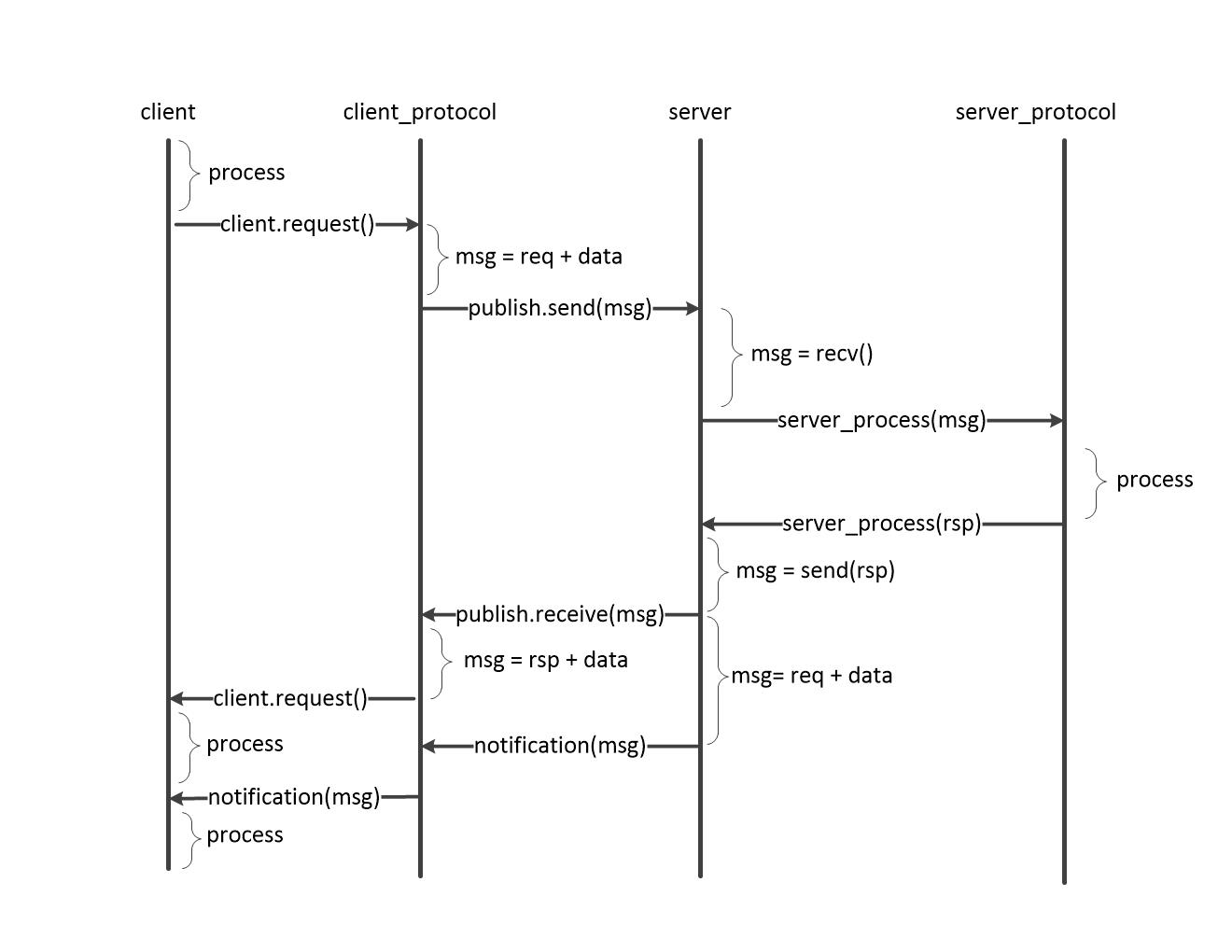
Implementation of Sudoku Application

1. Graphs & Figures
2. Use Cases Diagram



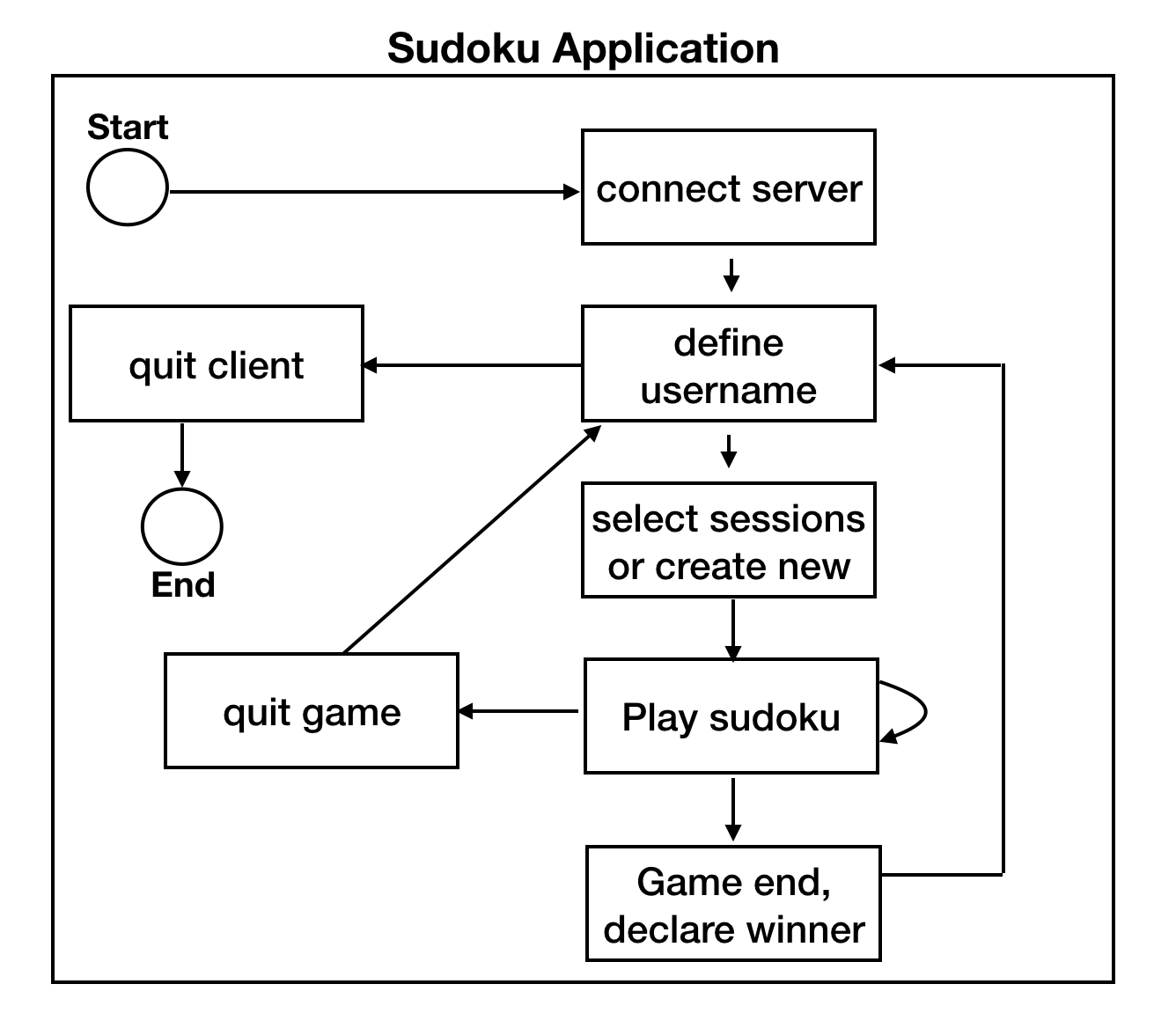
User-cases are as followed: user must make a connection, provide his/her name, and then he has 2 options a)create new game or b)join the game and In both cases user can leave the game. During game user makes moves and changes Sudoku.

1. Sequence Diagram



Sequence starts when the client’s main program calls for the request function. This function combines request header and data to create the message. Then, client protocol sends the message by publish function. Server receives the message with server\_process function and passes it to the server’s protocol. Server’s protocol extracts the message and evaluates the request. Then it sends the appropriate response via server\_process function. Client’s protocol receives the response and passes it to the client’s main. Also, server can use notification thread to notify the client according to the request type.

1. Block Diagram



The figure above is the block diagram for our Sudoku game. First, we start the program, we need to connect to the server. After connected, we setup a user name. This name is used in the game session and it’s a unique name in server. Then we can select to join an existing game or create a new game. After that, we should be in game session, we make moves until game finished or we can quit while in game session. Either quit game or game finished will lead us back to setup user name page. We can close clients there or continue to a new game.

1. Application details
2. Server Main

At first server socket is created, then it has been bound to the previously defined server address and server port. Server is in the listening mode and it is waiting for new client connections. For each client new thread is created by the server. Each thread is responsible to handle receiving messages from the client, provide it to the server protocol, and send all the requested information back to the client.

Communication with client is defined in “send\_receive” function. This function takes client’s socket, server’s socket and list of threads as arguments. At first server tries to take message from the client, if there is some error and message did not arrive, server will disconnect the client. After receiving message, the message must be processed by protocol so server main calls the function of server protocol: “server\_process”. Server main provides the protocol with message and client socket as arguments. And takes response and two types of notification threads. The response must be sent to that exact client. If response contains word “close”, it means that client wants to quit the game, so in that case server sends “OK” to the client and closes the socket. In some case it is necessary to provide some information to all the players of the specific game. In that case server main gets two types of notification threads and start them.

1. Protocol

Server protocol’s main function - Server Process deals with the messages received from the server. Firstly, we defined following requests in the beginning of the implementation:

* **Register new user** – Checks if there already exists the user with the provided name, if not it registers new user and responses okay, otherwise if the name is duplicated, it sends the response duplicated name.
* **Join the game**- here two different situations may appear a) user wants to join existed game or b) user wants to create new game. In case of a, protocol checks if the limit is not reached in the game session user wants to join, if is not reached it registers the user in the game session and makes user’s score zero, also notifies other users that new player joined the game, or if the limit is full sends notification about starting the game. Notification part is done by creating thread which calls notification-thread function, after this thread is sent to the server, server starts the thread. If limit is reached it responses –Limit already reached. In case of b, new Sudoku is generated, mode is depended on the user’s wish : easy, difficult or hard. The answers for this generated Sudoku is saved in the dictionary where the keys are game sessions. In the game dictionary each key is game sessions’ id which corresponds to another user dictionary, where keys are the user names and values are users’ scores and client’s socket. When new game is created user’s score is set to zero. After this response okay is sent.
* **Request for Sudoku**- all the existing game sessions with their Sudoku, limits and number of players are fetched and sent to the server.
* **Request for users**- user scores and names to the corresponded game-id are sent to the server.
* **Request for quitting from game**- In this case user wants to quit so he/she will be deleted with the whole information like name and score. All other users will be notified again by creating the thread and calling thread function. Thread will be sent to the server and it will be started there. In case all the users are quitted from the game, such game will be deleted. Client’s socket will be only closed when client sends the message ‘close’.
* **Request for move** – protocol checks if the position, in which the user wants to put the number is empty, if it is empty then checks if the number is correct, and increases user’s score by 1 in case of correct guess or decreases by 1 in case of wrong move. In case of correct move protocol also puts new number in the Sudoku and then checks if the Sudoku is full or not, in case it’s full it finds the winner and notifies all the users about it, if it is not full protocol sends 2 notifications about updating the score and Sudoku, this means it creates 2 threads, calls notification function inside the threads and sends to the server. In case of wrong move score is decreased and notification is sent about it. In case the position was not empty in Sudoku, protocol responses late move.

We arranged notification part by creating notification requests and notification-thread function. Notification-thread sends the notification to all clients, but thread is started by the server not in the protocol. Notification requests are:

* **Notify**- fetches user scores and names , to the corresponded game id and returns that message in string.
* **Sudoku1**- takes game-Id as parameter and finds Sudoku to that game id and returns it.
* **Winner**- takes game id as parameter, finds maximum score in that game session and finds the winner.
* **Startgame**- game session is started.

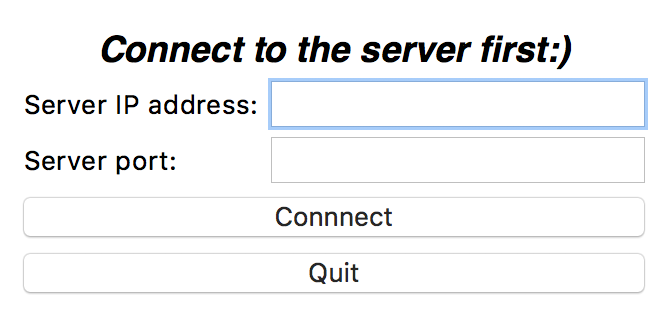
1. Client Main

In client side, we use the same protocol as server uses. We send requests with different headers and server will recognize the headers and process our message and give us response with response headers. Protocols are described in the protocol section.

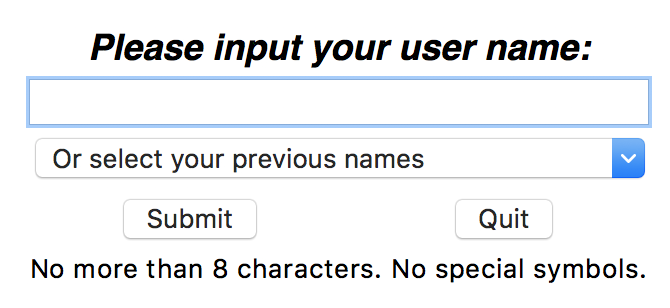
Tkinter is a python standard GUI package and most common used one. It’s a thin object-oriented layer on top of TK.

I create a class inherit TK class, and in this main class, I add different frames (also classes) in it, so it can load all frames when started and raise one when I need to change. There are five frames in total, ‘ConnectServer’, ‘Login’, ‘Joining’, ‘Newsession’, ‘Gamesession’.

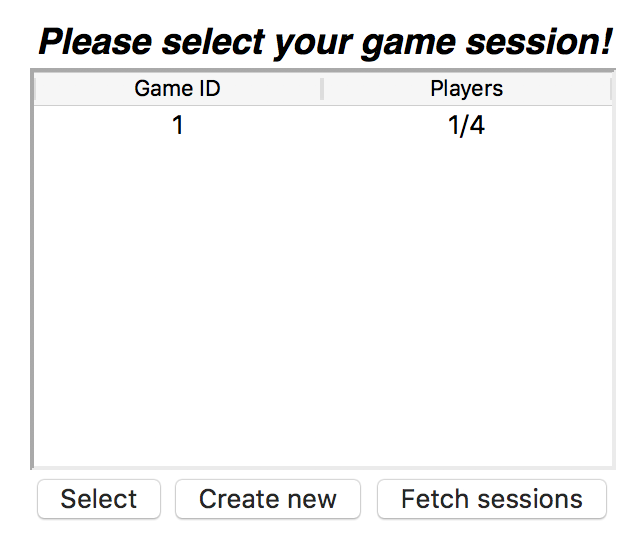
‘ConnectServer’ is the first frame when we open the client. We can see the graph below, there is a form we need to fill in to connect to server, or there is a quit button to close the client. After we filled in the server address and port then click connect, we will get into the second frame.



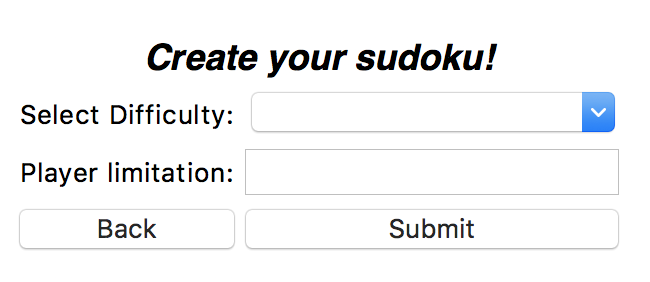
‘Login’ is the second frame where we can setup a username. As we can see the graph below, here we define the name. We can create a new name or select the previous name. (previous name only exists when you play the second or more times without closing client, to be precise, after 60 seconds as I defined in the client). The name also has limitations, it can’t be too long and no special characters. Then we can submit it to the server we connected before, server will check it if is unique name, if no problem, we’ll get into the third frame. Or we can quit client here also, that will cut down the connection with server.



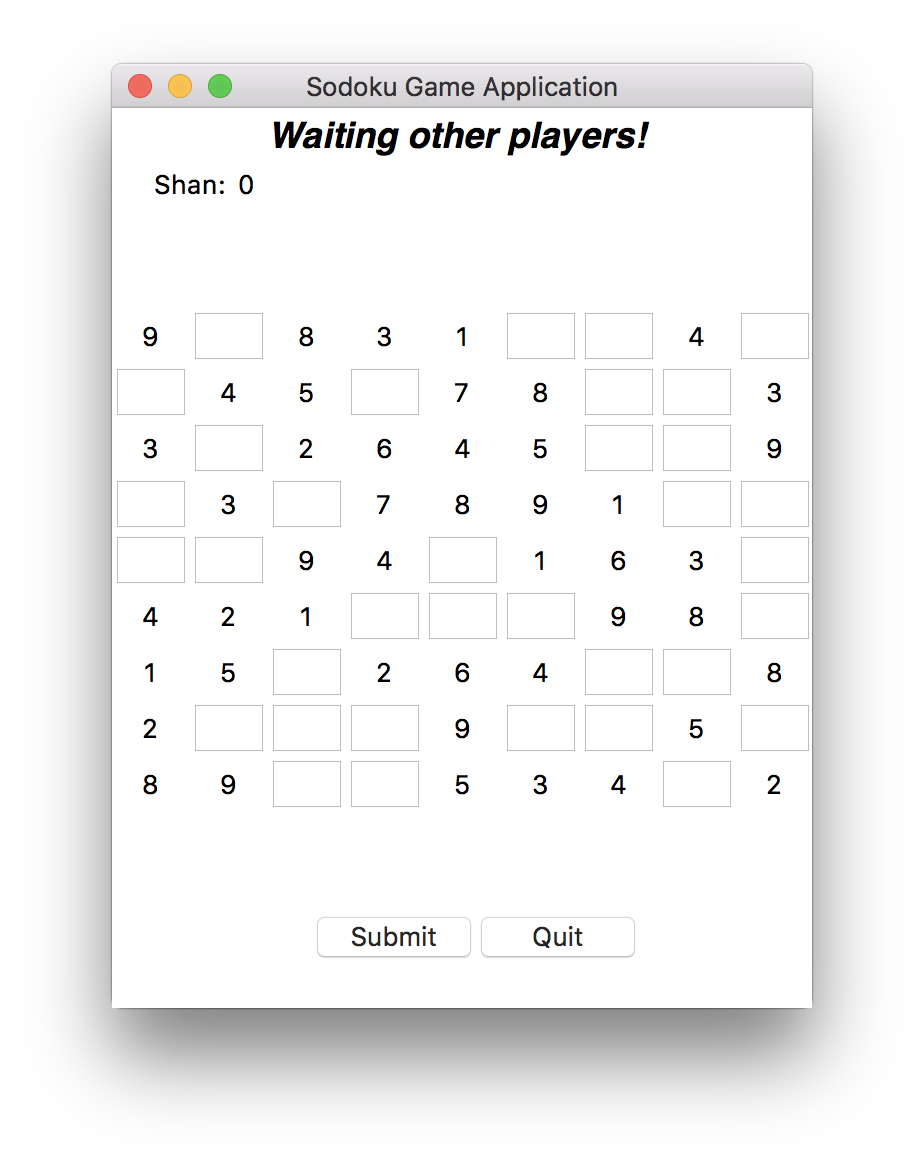
‘Joining’ is the third frame where we can select game session or decide to create a new one. As we can see from the graph below, there is a Treeview and three buttons. Treeview will display all the sessions we have in server, it will refresh automatically (as I defined is 30 seconds). In first column, it will show the game ID, it’s in ascending order and kept in server to distinguish different sessions. The second column is players numbers for user to browse which one has empty slot. ‘1/4’ means there is one player in the session while the limit number is four. After selected the session want to join, we can click select button to get into the final frame. Server will check if the session is available before join. But if we want to create a new session, we can select ‘create new’ button, and this lead us to the fourth frame.



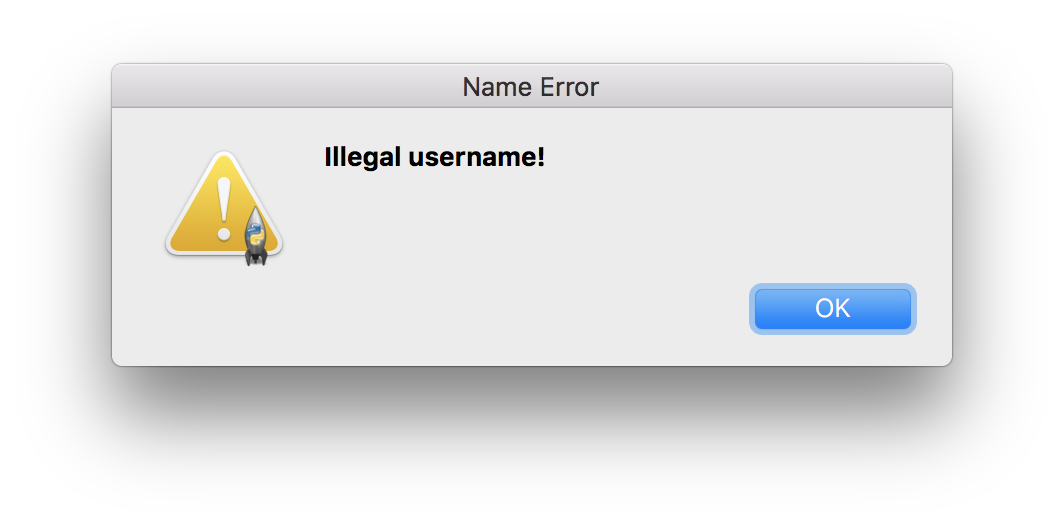
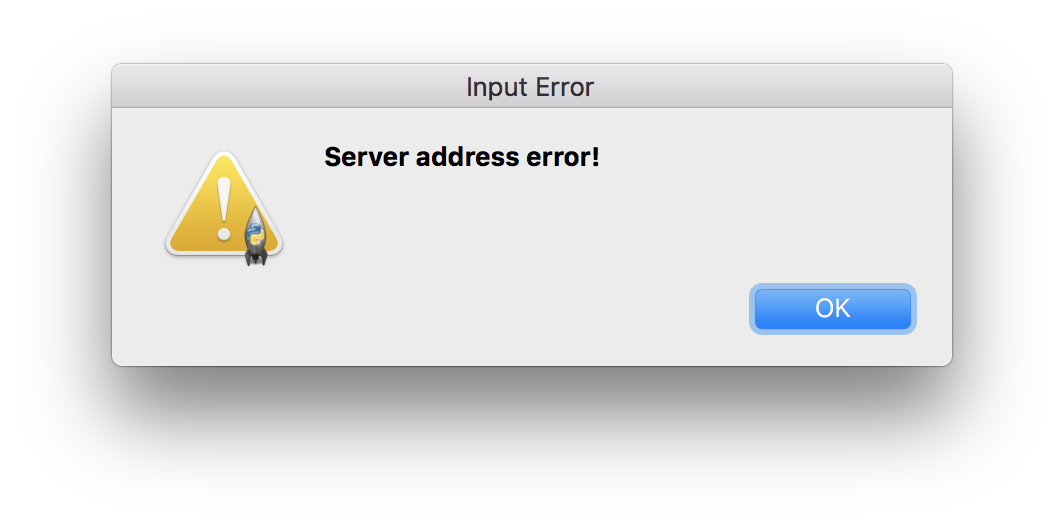
‘Newsession’ frame only displayed when a user wants to create his own game. We can select the difficulties of the game, easy, medium or hard. And we must set the limit number of players. Server will create a game session based on this two information we provided. After create the game, we will go to the final frame, or we can choose to go back to join others’ game.



‘Gamesession’ frame is where we play the game with other players. Players in the game session will wait until all the players are here. If game is full, game will start. Then we can make a move by filling a blank entry and click submit. (we can’t fill in two fields in one submission). Client will also be notified when other players make a move, it will update the score board and Sudoku. After the game end, client will declare the winner and score, then back to the second frame ‘Login’. User can also choose to quit while in the game session, this will clear the scores.



The client also can handle some kind of errors. Here is some example of type and operation errors.





1. Problems

Our homework still has some little problems. First, we do start from protocol, but we didn’t define all functions and design details. That makes us suffering changing of codes. Then we changed our plan, figured out all operations and communication details, that made us can work in parallel.

But our code still need to debug… We do finish the coding but we lack time to debugging it. The application is partly work. If we have time, we’ll continue finish it until make all functions described above work.

We believe next homework will not be so messy.