Tic Tac Toe Design Document

XML-RPC, Python

SWE 545 Distributed Systems, Boğaziçi University

Suzan Ece Ada

INTRODUCTION

This is a design document of a multi-client Tic-Tac-Toe game written in Python by the XML-RPC library. The primary requirement of the game is to allow multiple clients to access the server and play tictactoe with the computer simultaneously. In order to satisfy this requirement division of the engine, server and client structure was necessary. Therefore 3 python files should be created at the start one being Game.py and the others being tictactoe\_client.py, tictactoe\_server.py.

First task should be designing the TicTacToe engine. Because this is an XML-RPC Server and Client engine extra attention should be given to using proper statements. Input from the user will be considered as arguments to the TicTacToe function. Subsequently a parser should be designed in the server for the Game.py engine, to create TicTacToe session instances and implementing XML-RPC methods.I designed the TicTacToe engine and make sure the computer starts at the corners in order to increase the computer’s chance of winning.

After making sure that the engine, client and server work fluently as a coherent structure we will be tackling the issue of optimising server architecture to handle multiple clients simultaneously.

XML-RPC API DOCUMENTATION

*def generateClientID():*

This function generates a unique sessionID in the system and works as a key to access the Server. It returns an integer sessionID which determines the sessionID of the client.

*class Parser():*

*def \_\_init\_\_(self):*

*self.clientDict = {}*

Above function shows the initialisation of the Parser class. As soon as the instance is created an empty Client Dictionary is created also. Each time a client generates a Client ID and starts a game a new game instance will be added to this dictionary. The value part of each dictionary element is a Game instance whereas the key is the unique sessionID generated.Below functions are all methods of the above class.This part will not be used in the Client side. This is here for explanatory purposes.

*def setSessionID(self,sessionID):*

This function is used to create a Game instance in the server with the given SessionID. It doesn’t return anything.Its purpose is to make sure a Game instance of the sessionID is added to the dictionary.

*def startGame(self,sessionID):*

This function returns the welcoming string of the game as well as instructions. Its only used. It takes the sessionID that is generated in the first function as the only argument.

*def printBoard(self,sessionID):*

This function returns the current overview of the board as a string. It again takes the unique sessionID as the only argument.This should be used whenever a move is made and the current position of the symbols have changed.

*def getGameStatus(self,sessionID):*

*This function takes the unique sessionID as the only argument. It returns the current gameStatus of the client. It will return a string. “0” means that no one has won and the game should continue. “1” means that the computer has won. “2” means that the game player has won and anything else means that there is a tie. Because the gameStatus is defined at the search engine internally and no other input is assigned to it there is no need for error check.*

*def checkMove(self,sessionID):*

This function takes the unique sessionID as the only argument. It doesn’t return anything but it changes adds a new “X” symbol to the board. This is the movement of the computer as a reaction to the current board state.

*def checkWinner(self,sessionID):*

This function takes the unique sessionID as the only argument. It just checks whether the game ended or not. It returns True if the game ended thus there is a winner or there is a tie. It returns False if game is still going on.

*def checkValidMove(self,sessionID,playerCell):*

This function takes the unique sessionID as the first and the number that the client played as String as the second argument. It returns True if the move is Valid or returns False if the move is invalid and that place is either occupied or a wrong number is given as the second argument.

*def movePlayer(self,sessionID,playerCell):*

This function takes the unique sessionID as the first and the number that the client played as String as the second argument. It doesn’t return anything but changes the look of the board. It might also change the state of the game if the user wins. Player’s move should be put in this function as the second argument after the validity of the move is tested by the previous function.

*def strGameStatus(self,sessionID):*

*This function takes the unique sessionID as the first argument. It returns the string explanation of who won if the game ends. If the gameStatus is currently “0” and the game continues it doesn’t return anything. However if the game has ended and the winner should be announced or the result of a tie should be informed this function returns the String description of how the game ended.*

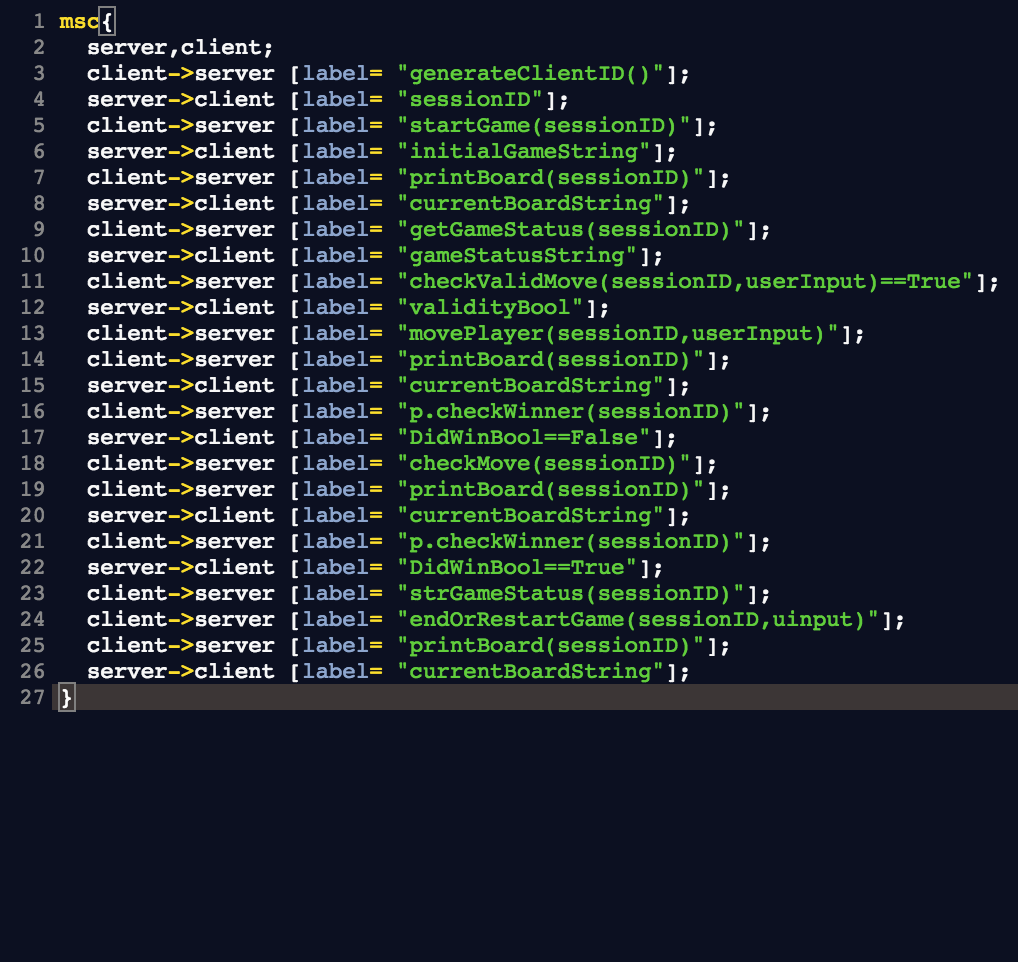
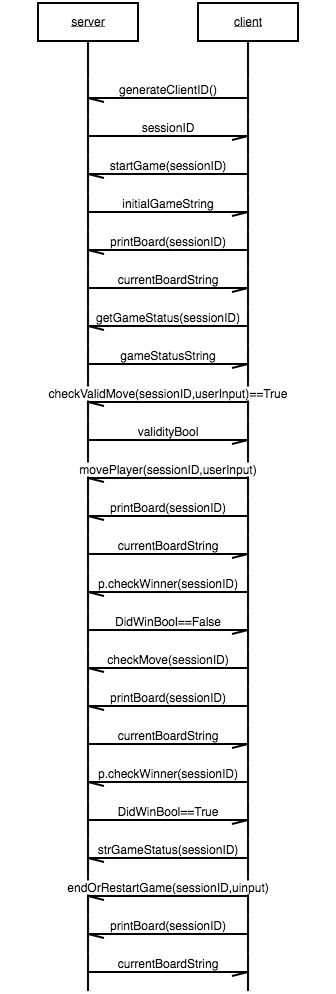
*def endOrRestartGame(self,sessionID,uinput):*

This function takes the unique sessionID as the first and the input character of the client as the second argument. It resets the board and generates computer’s first move based on the uinput string that is given as the second argument. If the second argument is “y” it restarts the game by just restructuring the board, if something else it returns nothing.

*def removeSession(self,sessionID):*

This function takes sessionID as the argument and removes the Game instance from the dictionary so that the games that have ended or quitted are removed.

MESSAGE FLOW SCENARIOS



Above is a message flow scenario between client and server. This sequence chart is generated online by the Mscgen. Link of the site is <https://mscgen.js.org/>.

As seen on the chart some boolean values are given in order to illustrate the flow in those cases.

HANDLING MULTIPLE GAMES SIMULTANEOUSLY

In order to handle multiple games an empty dictionary is initialised when the Parser class is registered as an instance. There is a static counter in the server which increases by 1 each time a client tries to play with it by the generateClientID function. Therefore each time a client requests to play a game a Game instance is initialised with that generated Client ID (increased counter number in the server) and added to the clientDict{} in the server. Subsequently functions that are implemented from the client are implemented on the clientDict element corresponding to that sessionID. Because this is a quick game where small data transfer occurs no extra thread is used in the process and each client gets its response nearly simultaneously. In addition when if the client hasn’t responded for 5 minutes server side automatically disconnects the client and removes the Game instance which corresponds to that sessionID from the clientDict. Again the exited clients are removed from the Server to prevent unnecessary memory usage.

The key data structures used in this project are strings, boolean, integers, array and dictionary. Strings to print the Board to allow the user to visualise the game.Booleans are used in order to check the movement validity or whether the game has ended or not. Dictionary is used to map sessionIDs to the relevant Game instances to allow multiple player to play the game simultaneously.

If the Client doesn’t respond for more than 5 minutes we assume that the client lost interest in the game or better couldn’t come up with an answer in that limited time. In this case the client is told that he/she has lost and the connection is automatically shut down.

CONCLUSION

Some minor bug fixes are made and a timer is added to illustrate a more entertaining game and get rid of unneccessary load on server. This project gave me a better understanding of Python, client–server architecture and distributed systems after applying the cumulation of knowledge I gathered from research and examples.