

TIC TAC TOE Game Using RPC-XML

Server – Client Architecture Implementation

SWE 545 Distributed Systems Programming

Term Project

**Name :** İlker YILMAZ

**Student ID :** 201571915

**Introduction**

XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism. XML-RPC works by sending an HTTP request to a server implementing the protocol. The client in that case is typically software wanting to call a single method of a remote system. Multiple input parameters can be passed to the remote method, one return value is returned. The parameter types allow nesting of parameters into maps and lists, thus larger structures can be transported. Therefore, XML-RPC can be used to transport objects or structures both as input and as output parameters.

The goal of this project, is to implement a “tic tac toe” game using Python’s SimpleXMLRPCServer (server-side) and xmlrpclib (client-side) modules with any “tic tac toe engine from internet and thoroughly comprehend the ideas that we’ve learnt through this course.

In this respect I’ve used a simple tic tac toe engine from github <https://github.com/cklone/tictactoe-python> and I’ve modified it to fit in my purposes. The complete project files can be found in my github folder at <https://github.com/iy536/SWE545termproject>.

### Server Internals and API Documentation

server = **xmlrpclib.ServerProxy**(**"http://localhost:8000"**)

Establishes a connection to game server

*def* ***create\_game****(self):*

This function initializes the TicTacToe game engine with 3x3 board size. At the same time, when this API is called, it creates a new thread and mapped it with a unique client id. If a client wants to start a game, client should first call this function. This returns a unique client id clientId

*def* ***start\_game****(self,clientId):*

Starts created game with associated clientId, when **print** **start\_game(**clientId**)** is called, the welcome banner of the game is displayed on client screen

*def* ***print\_board(****self,clientId****)***

This function displays the current state of the tic tac toe board with associated client ID. It should be called each time when player makes move.

e.g

|O |X |X| |1 |2 |3 |

| |O |X| |4 |5 |6 |

|O |X |O| |7 |8 |9 |

*def* ***make\_move(****clientId,reply****)***

This function sends the user input which is given by **reply** argument [1 to 9] to tic tac toe server. **reply** argument corresponds to position in the board where user wants to make a movement. Server receives given input from client with this command and make counter move. Then this function returns if the game is ended or it is still being played.

*def* ***move\_valid(****clientId****,****reply****)***

This function checks the user input, which is called by ***make\_move(****clientId,reply)*, is valid or not. If the tic tac toe board size is 3, the valid input is between 1-9. Also, If the cell already is filled, function returns false and asks user to enter a move until it is valid.

*def* ***print\_result(****clientId****)***

After the game is finished which is decided after make\_move(clientId,reply) function, this function must be called to display the result of the game. It returns the following banner strings respectively in the case of game status.

* If the user wins, displays "Winner" banner
* If the computer wins, displays "Loser" banner
* If the game ties, displays "Game Tied"

*def* ***end\_game(****clientId****)***

When this function is called, server kills the thread which is attached to this client and end the associated game.

### Use-Case Scenarios Message Flow

### The following diagram shows message flow between client and server.

### 

### Figure 1 – Message Flow

### Design Technicality – Handling of Multiple Games Simultaneously

### Server is able to serve multiple clients simultaneously. To achieve this, when *TicTacToeGame()* class is registered as an instance, an empty *clientMapping* dictionary and a *clientId* is being created. Each time a game play requested by a different client with *create\_game()* function, a unique *clientId* will be assigned to *game()* object and a thread is being created for each *game()* object with this *clientId*. Every clientId is mapped to its own thread, so server can serve multiple clients using this unique *clientId*.

### Whenever a function called by client, server matches its *clientId* with the corresponding thread and completes function calls. Figure 2 shows a simple visualization of threaded game play.

### 

### Figure 2 – Mapped Threads

### Data Structures

### Integer data structure is used to give clientId for each client, and as arguments passed to functions

### Thread mechanism is used to enable multiple games simultaneously.

### Dictionary structure is used to map clientId and associated Thread

### String data structures are used for function returns and also print the result banners

### Boolean data structures are used to keep track of the game status (playing or ended)

### List data structure is used in original game.py which is used as tic tac toe engine

### game() class is used as Tic Tac Toe game engine which is found in github

### TicTacToeThread() class is used to pass each clients requests to its game object

### TicTacToeGame() class manages all TicTacToeThreads’

### Conclusion

### After completion of this project assignment, I have had deep knowledge on xml-rpc protocol, understanding of threading mechanism, client-server architectures and distributed systems. Also I have had a lot of experience on Python programming. I believe project served its goals well.