Names: Jake Tusa, Michael Cheung, Billy Ko

Date: 5/6/17

CSE 310 Final Project Server Documentation

**TASKS COMPLETED:** Parts 1 and 2

========================**USER DOCUMENTATION**=======================

**OBJECTIVE OF REVERSE TIC-TAC-TOE**

The game has two players and involves a 3x3 board. One player will use ‘X’ pieces, while the other uses ‘O’ pieces. The two players will alternate and place their pieces onto the board one at a time. The first player to place three pieces in a straight line loses the match. If there are no more available spots on the board but neither player has three pieces in a straight line, then the match is a draw.

**SETTING UP A SERVER**

**INSTRUCTIONS**

1. Run server.py in a command line

**RUNNING THE CLIENT**

**COMMANDS**

help this command takes no argument. It prints a list of supported commands, which are ones in this list. For each command, it prints a brief description of the command function and the syntax of usage.

login this command takes one argument, your name. A player name is a userid that uniquely identifies a player. Your name is entered with this command and is sent to the server.

place this command issues a move. It takes one argument n, which is between 1 and 9 inclusive. It identifies a cell that the player chooses to occupy at this move.

exit the player exits the server. It takes no argument.

games this command triggers a query sent to the server. A list of current ongoing games is returned. For each game, the game ID and game players are listed.

who this command has no argument. It triggers a query message that is sent to the server; a list of players who are currently logged-in and available to play is retrieved and displayed.

play this command takes one argument, the name of a player X you'd like to play a game with.

**INSTRUCTIONS**

1. Run client.py in a command line with two additional arguments:

1. The name of the machine running the server

2. The port number that the server is listening at

If the machine you are trying to connect to is not running the server or does not exist, or if the port number is incorrect, then you will receive an error message with the reason for the error.

2. After connecting to the client, use the “login” command followed by a username. If the username is not already taken, then it will become your username. Otherwise, the server will return an error message stating that the username is unavailable and the user must use the login command again. Using the “place” command at this stage will return an error message and nothing else will happen.

3. Once you have logged in, the client will then wait until the server has two players to begin a new match of reversed tic-tac-toe. Once the server finds two players, a new match of reverse tic-tac-toe will automatically begin, and your client will display the tic- tac-toe board as well as the opponent’s username.

At any time, you may use the command ‘exit’ to exit the server and log out, ending any active matches.

4. You will be notified via command line when it is your turn. During your turn, use the “place” command followed by the position on the board that you wish to place your piece on. If the position is already occupied, or the number you entered is invalid, you will receive an error message and must use the “place” command again to make your move. Using the “place” command when it is not your turn will return an error and nothing else will happen. After successfully making a move, the client will display the updated tic-tac-toe board and you must wait for the other player to make his/her move.

5. Once one player has 3 pieces in a row or if the board is fully occupied, the client will display whether you won, lost, or drew. You will then automatically wait for another match to begin.

6. To stop playing, use the “exit” command.

=======================**SYSTEM DOCUMENTATION**======================

**REVERSE TIC-TAC-TOE PROTOCOL**

**METHODS:**

**210 LOGIN**

The LOGIN method means an attempt to for a client to log into the server. The contents of the request includes an ID set by the user.

**211 PLACE**

The PLACE method means an attempt for a client to place a piece on the tick-tac-toe board. The contents of the request includes the position on the board in which the player wishes to place a piece on.

**212 EXIT**

The EXIT method means that a client has exited from the game.

**222 WHO**

The WHO method means a request for the IDs of all available players connected to the server who are not in game.

**223 GAMES**

The GAMES method means a request for all active game IDs and the IDs of its players.

**224 PLAY**

The PLAY method means a request to play with a specific player. The contents of the request includes the ID of the target player.

**STATUS CODES:**

**200 OK**

The request has succeeded. The information returned with the response is dependent on the method used in the request, for example:

PLACE the status of the newly updated tick-tac-toe board

**213 WAIT**

The WAIT status code is used for two scenarios:

1. The login request is successful, but there are not enough players to begin a match. The client must wait for a match to begin.

2. The client is participating in a match, but it is the other player's turn to make a move. The client must wait for the other client to make a move before it can make one.

**214 START**

There are enough players to begin a match, and a match has begun.

**215 GO**

It is the current player’s turn in the match

**216 WON**

The match has ended and the client has won

**217 LOST**

The match has ended and the client has lost

**218 TIED**

The match has ended and both clients tied

**219 NAME**

Contains the name of the opponent player

**220 LEFT**

The opponent player has left the match, and the match has ended.

**221 DISPLAY**

The game board has been updated. Contains the status of the game board.

**400 ERROR**

An invalid request was received or an error occurred executing a request.

**REVERSE TIC-TAC-TOE SERVER**

**SERVERP1.PY**

\*\*\* This is the server for Part 1 of the final project

**LANGUAGE**

Python 3.5

**MODULES**

socketserver used to simplify the task of writing network servers by server objects with handlers and stored client information

threading used to create multiple threads to multiple clients simultaneously

time used for the sleep method which pauses a thread

**GLOBAL VARIABLES**

The server class contains global variables for each of the protocol methods and status codes, which can be found in section 1.1 and 1.2 in this documentation, as well as local variables:

connections a counter of the number of connected clients

playerList a list of player objects representing the current players

nameList a list of IDs of current players; used to check if an ID is available for login

game a Game object to represent status of the match

**CLASSES**

**ThreadedTCPHandler** a handler object which contains a handle() function that does all the work required to service a request

**ThreadedTCPServer** a server object that uses a handler object to handle all requests to the server

**Player** an object to represent each player

**Game** an object to represent an ongoing match

**ThreadedTCPHandler(socketserver.BaseRequestHandler)**

**FUNCTIONS**

**handle(self)** handles all requests from the client to the server

**ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer)**

Inherits the functions and variables from both socketserver.ThreadingMixIn and socketserver.TCPServer

**Player**

**VARIABLES**

name name of player

state represents whether the player is busy or available

piece shape of the player’s piece, ‘X’ or ‘O’

isTurn flag to indicate if it is the player’s turn

playerWaiting flag to indicate when a player is waiting for his/her turn

playerExited flag to indicate if a player has left the match

**METHODS**

init class constructor that automatically runs when instantiated

getName returns name

getState returns state

getPiece returns piece

getIsTurn returns isTurn

setName sets name

setState sets state

setPiece sets piece

setIsTurn sets isTurn

**Game**

**VARIABLES**

NUM\_PLACES static variable for the number of positions on the board

BLANK symbol for an available position on the board

TIE return value when a game ties

playerList list of players in the match

gameBoard list with each index representing a position on the board

isActive flag to indicate that the match is active

**METHODS**

getPlayerList returns playerList

getIsActive returns isActive

setisActive sets isActive

addPlayer adds a player to PlayerList

removePlayer removes a player from PlayList

createBoard initiates gameboard

displayBoard returns a string visualization of the current game board

updateBoard updates the game board with a new piece

checkLoser checks board for losing or tie condition

**CODE**

**handle(self)**

**Variables**

The function references all of the global variables that need to be shared.

“killThread” is a variable used for control flow that is checked when a player exits

**Sleeping threads**

Before each time the server sends a message to the client, it calls “sleep” to pause the thread to avoid race conditions or sending too many messages too clients at once.

**Accepting incoming connections**

If there are less than 2 connections, then upon accepting a connection, the server prints a message and sends an “OK” protocol message to the connected client.

If there are already 2 connections, then the connecting client will receive an “ERROR” message.

**Handling commands or messages before a game has begun**

A variable “loginSuccess = False” is used as a flag to indicate if a player has logged into the server and is set True when a player successfully logs in.

A while loop is executed while loginSuccess is false, in which the server receives messages from the client.

If the message is a “LOGIN” request, then the server checks if the name is available and returns “OK” to the client if it is, and “ERROR” if it isn’t.

If the message is an “EXIT” message, then the server returns an “OK” message and sets “killThread = True” and exits the function.

For all other requests or messages, the server returns an “ERROR” message to the client.

After a player logs in, if there are no players in the playerList, then a player object is created with “player = Player(name, "available", "X", True)”, and assigning the ‘X’ symbol to that player. If there is already a player in the playerList, then a player object is created and the ‘O’ symbol is assigned to the player.

If there is only one player connected, a “WAIT” message is sent to the player.

**Starting a game**

Once a second player is connected, a game object is instantiated and the two players are added to the game’s playerList, and both players’ states are set to ‘busy’ and the second player’s PlayerWaiting is set to true to indicate that it is the other player’s turn.

The server sends a “START” message to indicate that a match has started as well as a “NAME” message with the name of the opposing player to each of the two players.

playerExited is set to False to indicate that no player has previously left that match, and the game’s isActive is set True.

**While a game is ongoing**

A while loop runs while isActive is True. At the start of an iteration of the loop, if killThread is true (due to a player leaving), the loop stops and exits.

If “playerExited = True”, indicating that a player left the game, a “LEFT” message is sent to the remaining player and sets the player to be first with ‘X’ pieces and waits for a second client to connect.

The server sends a “DISPLAY” message with a visualization of the board to both of the clients and calls “game.checkLoser()” to see if the losing or tying condition has been met. If it has, the server will send the appropriate protocol message to each of the clients and then restarts the game.

Otherwise, the server checks which player’s turn it is and sends a “GO” message to that player.

**Handling a ‘login’ command during a game**

If the server receives a “LOGIN” request during a game, it ignores the request and replies with an “ERROR” message.

**Handling a ‘place’ command during a game**

If the server receives a “PLACE” message during a game and it is not the sending client’s turn, it is ignored and the server replies with an “ERROR” message.

If the server receives a “PLACE” message from the client whose turn it is and the position is invalid, the server replies with an “ERROR” message. If the position is valid, the server updates the game board, responds with an “OK” message, and sets the turn to the other player’s.

**Handling an ‘exit’ command during a game**

If the server receives an “EXIT” message during a game, the server sends an “OK” message back to the client that sent it, removes the player from the playerList and the game object, sets killThread to True to indicate that a player has left, and decrements the number of connections.

**Handling all other messages or commands during a game**

If the server receives a message that is not any of the previously mentioned commands, it sends an “ERROR” message back to the client that sent it.

**Handling if the other player has left the game**

If the other player has left the game, the remaining player is put back into a loop waiting for another player to connect.

**CONTROL FLOW**

**Initializing server**

The server has two static variables for the hostname and port.

The server then passes these two variables as arguments to construct a ThreadedTCPServer object that creates a ThreadedTCPHandler each time a client connects to the server.

The server then uses “server\_thread = threading.Thread(target=server.serve\_forever)” to enable multithreading and “server\_thread.start()” to begin multithreading.

**Receiving a connection**

Upon receiving a connection from a client to the localhost port 1337, ThreadedTCPServer creates a new thread associated with the client and creates an instance of the ThreadedTCPHandler object. The thread runs the “handle(self)” method from the ThreadedTCPHandler object, which handles all the messages received from the client.

**Terminating connections**

If a player exits, the client’s associated thread will update the game status in the “handler” function and then terminates the thread.

**SERVERP2.PY**

\*\*\* This is the server for Part 2 of the final project. It shares many of the same assets as serverp1.py, but has the following differences:

**GLOBAL VARIABLES**

totalGames a count of how many active games have been started on the server

**CLASSES**

**ThreadedTCPHandler**

**FUNCTIONS**

**findGameByID(self, gameID)**

**Player**

**VARIABLES**

connSocket the connection socket the client used to connect to the server

**METHODS**

getConnSocket returns connSocket

setConnSocket sets connSocket

**Game**

**VARIABLES**

gameID ID for the match

**METHODS**

getGameID returns gameID

setGameID sets gameID

**CODE**

**handle(self)**

**Variables**

localGameID ID for the game running on the thread

lobbyLoop flag for controlling whether a client is in the lobby

exitLobbyLoop flag to allow a client to exit the lobby

**Accepting incoming connections**

Unlike in serverp1.py, the server does not prevent more than two clients from connecting and allows an indefinite amount of connections.

**Handling commands or messages before a game has begun**

If a client has connected but has not logged in and sends a “WHO” request to the server, the server responds with an “OK” message that contains a list of all connected players’ names.

If a client has connected but has not logged in and sends a “GAME” request to the server, the server responds with an “OK” message that contains a list of all active game IDs.

If a client has connected but has not logged in and sends a “PLAY” request to the server, the server responds with an “ERROR” message.

Unlike serverp2.py, The server no longer automatically puts two players into a match.

Instead, each thread for each client enters a loop “while lobbyLoop == True:” which simulates a lobby and during which handles messages from the clients.

**Handling a ‘login’ command in the lobby**

If the server receives a “LOGIN” request from a client in the lobby loop, it ignores the request and replies with an “ERROR” message.

**Handling a ‘place’ command in the lobby**

If the server receives a “PLACE” request from a client in the lobby loop, it ignores the request and replies with an “ERROR” message.

**Handling an ‘exit’ command in the lobby**

If the server receives an “EXIT” message from a client in the lobby loop, it replies with an “OK” message and terminates the thread containing the handler.

**Handling a ‘who’ command in the lobby**

If the server receives a “WHO” request from a client in the lobby loop, it replies with an “OK” message with a string containing all of the logged in players’ names separated by spaces.

**Handling a ‘games’ command in the lobby**

If the server receives a “GAMES” request from a client in the lobby loop, it replies with an “OK” message with a string containing all the active games’ IDs and their players’ names.

**Handling a ‘play’ command in the lobby**

If the server receives a “PLAY” request with the target player name from a client in the lobby loop, it checks if a player exists with the given name. If it does not exist, it responds with an “ERROR” message. Otherwise, the client that sent the request has its lobbyLoop set to False, letting it exit the lobby loop, sends an “OK” message to the client, and sends a “MATCHED” message to the target player. The server then sets the requesting player’s piece to ‘X’ and is its turn.

**Handling an ‘OK’ message in the lobby**

If the server receives an “OK” message from a client in the lobby loop, then a player has received a “MATCHED” message after another player sent a “PLAY” request. The player’s piece is set to ‘X’ and it is not its turn, and sets its lobbyLoop to False to let it exit the lobby loop.

**Starting a game**

After two players have been matched, a game object and its assets are instantiated, and the global totalGames variable is incremented. The game’s ID is then set to totalGames’s value. Since totalGames is never decremented, no two games will have the same ID.

The server then sends the game’s ID as well as the opposing player’s name to both players.

**Handling if another player left the game**

If an opposing player left the match, then the remaining client will be put back into the lobby loop.

**Handling a ‘who’ command during a game**

If the server receives a “WHO” request from a client in a game, it replies with an “OK” message with a string containing all of the logged in players’ names separated by spaces.

**Handling a ‘games’ command during a game**

If the server receives a “GAMES” request from a client in a game, it replies with an “OK” message with a string containing all the active games’ IDs and their players’ names.

**Handling a ‘play’ command during a game**

If the server receives a “PLAY” request from a client in a game, it ignores it and replies with an “ERROR” message.

**REVERSE TIC-TAC-TOE CLIENT**

**CLIENTP1.PY**

\*\*\* This is the client for Part 1 of the final project

**LANGUAGE**

Python 3.5

**MODULES**

sys used to obtain command line arguments at compilation

socket used to connect the client to a server socket to exchange messages

**GLOBAL VARIABLES**

The client contains a global variable for each of the protocol methods and status codes, as well as two variables for the port and hostname:

PORT the port number that the server is listening at. Defaults to ‘1337’

loggedIn a flag to indicate whether the client is logged into a server. Defaults to False

**FUNCTIONS**

**main()** this is the main function of the client, and is the only function executed by the client

**CODE**

**main()**

**Checking the number of command line arguments**

“len(sys.argv)” returns the number of arguments passed into the command line at run time. The python script must be run with 3 arguments: the python file name, the hostname of the computer running the server program, and the port the server is listening at. Any other number of arguments will print an error and terminate the program.

**Checking the validity of the command line arguments**

“sys.argv[1]” is the argument following the python file name. This is expected to be the hostname of the server.

“sys.argv[2]” is the second argument following the python file name. This is expected to be the port number that the server is listening at. In this implementation, the server’s port is always ‘1337’.

“try: portNumber = int(sys.argv[2])” converts the port number argument from a string to an integer. If the argument is not a number, then an error is thrown and caught by “except ValueError:”, which prints an error message and terminates the program. If the argument is not ‘1337’, then an error message is printed and the program is terminated.

“clientSocket = socket(AF\_INET, SOCK\_STREAM)” initializes the client socket that will be used to establish a connection with the server.

“clientSocket.connect((HOST, PORT))” connects the client socket to the server. If the connection is refused, the server hostname entered is invalid and an error message is printed and the program terminates.

**Connecting to the server**

“clientSocket.recv(1024).decode().split()” decodes a message received from the connected server and splits it into a list separated by spaces. If the first entry in the list is the protocol number for “OK”, the server has accepted the connection. Otherwise, the connection is rejected and the client program terminates.

**Main loop for handling input**

An indefinitely running loop that handles all input from the player as well as all messages received from the server.

“arguments = input("> ").split()” prints “> “ to the command line and waits for the user to enter a command into the command line, which is split into a list separated by spaces named ‘arguments’.

**Empty input**

If the input is empty, then the loop begins again. Otherwise, the loop checks the input to determine if it is a valid command and acts accordingly.

**Handling the ‘help’ command**

Checks if the input is “help” and prints the ‘help’ message if true.

**Handling the ‘login’ command**

Checks if the input is “login” followed by an argument (the username).

If true, the ‘login’ command is executed.

“loginMessage = LOGIN + " " + arguments[1]” creates a string named ‘loginMessage’ with the LOGIN protocol and the player’s username.

“clientSocket.send(loginMessage.encode())” encodes and sends the login protocol to the server.

“response = clientSocket.recv(1024).decode()” waits for and receives a response from the server, and then decodes the message.

“tokenized = response.split()” splits the decoded response message into a list separated by spaces. The first entry in the list is then checked to see if an “OK” protocol message is returned, which will then continue the loop, or if an “ERROR” protocol message is returned, which will print an error if the username is invalid or if the player is already logged in, and the loop will begin again.

If the message is an “OK” protocol message, then the global variable logedIn is set True and the client then waits for another response from the server.

If the next message received is a “WAIT” protocol message, then there are not enough players to begin a match and the client waits until it receives a “START” protocol message to indicate that a match has begun.

Once the “START” protocol message has been received, the client waits for a “NAME” protocol message containing the name of the opponent, and prints the name of the opponent once it receives the message.

Then the client then waits for a “DISPLAY” protocol message containing the contents of the board, after which it will notify the player that the game has begun and will print out the board and waits for the next response.

If the next response is a “GO” protocol message, the client will notify the user that it is his turn and the loop will restart to handle the user’s next input.

Otherwise, if the response is a “WAIT” protocol message, the client will notify the user that it is the other player’s turn.

The client waits for a “DISPLAY” protocol message and updates and prints the board upon receiving it and waits for the next response from the server.

If the next response is a “GO” protocol message, the loop will restart to handle the user’s next input.

If the next response is a “LEFT” protocol message, the opponent left the match and the client waits for another match to begin.

**Handling the ‘place’ command**

Checks if the input is “place” followed by an argument (the position on the board to place a game piece). If true, the ‘place’ command is executed.

“tileNumber = int(arguments[1])” converts the argument to an integer. If the argument is not a number, or is not a number in the range of 1 to 9, then an error is printed and the loop will restart to handle the user’s next input.

If the command is valid, the client uses “placeMessage = PLACE + " " + arguments[1]” to create a “PLACE” protocol message and “clientSocket.send(placeMessage.encode())” sends the message to the server.

If the return message is an “ERROR” protocol message, then the loop restarts to handle the user’s next input.

Otherwise, if the server returns an “OK” protocol message, the next response message received from the server contains the updated game board.

The client then waits for the next message from the server.

If the next message is a “WAIT” message, then the client informs the player that the move was made successfully and that it is the opponent’s turn. If the next message afterwards is a “LEFT” message, then the game has ended and the client waits for another game to start.

If the message is a “WON”, “LOST”, or “TIED” protocol message, then the game has ended and the client displays whether the player had won, lost, or tied, respectively, and waits for another game to start.

If the message is a “GO” message, the client will notify that it is the player’s turn and the loop will restart to handle the user’s next input.

**Handling the ‘game’ command**

Checks if the input is “games” with no following arguments. If true, the ‘game’ command is executed.

“gamesMessage = GAMES” creates a “GAMES” protocol message.

“clientSocket.send(gamesMessage.encode())” sends the protocol message to the server.

The client then waits for a response message from the server.

If the next message is an “OK” protocol message, it contains the data of the other games as a list of strings containing the game ID and the two player IDs separated by spaces. If the message only contains the protocol, then there are no active games. Otherwise, the client splits each of the strings in the list, separated by spaces, and prints them.

If the message is an “ERROR” protocol message, then the client prints an error and restarts the loop.

**Handling the ‘who’ command**

Checks if the input is “who” with no following arguments. If true, the ‘who’ command is executed.

The client sends a “WHO” protocol message to the server and waits for a response.

If the response is an “OK” message, it contains a list of the IDs of all players ready to play. The client then prints the IDs and restarts the loop.

If the response is not an “OK” message, then the client prints an error message and restarts the loop.

**Handling the ‘play’ command**

Checks if the input is “play” followed by an argument (the ID of the target player). If true, the ‘play’ command is executed.

The client sends a “PLAY” protocol message to the server and waits for a response.

If the response is an “ERROR” protocol message, then the client prints an error message and restarts the loop.

If the response is an “OK” message, then the server has created a game with player and the target player, and the client restarts the loop to handle the user’s next input.

**Handling all other inputs**

For all other inputs, the client prints an error message to indicate that the input is invalid, and the loop restarts.

**CONTROL FLOW**

**Running main()**

Calls the main() function on execution of the python file

**CLIENTP2.PY**

\*\*\*This is the client for Part 2 of the final project. Clientp2.py shares the same assets as clientp1.py, but has the following additions/changes:

**MODULES**

selectors used to listen and send out messages simultaneously

time used for the sleep function to put threads to sleep

**GLOBAL VARIABLES**

inGame flag to indicate if the client is in a game

clientSocket the socket that the client uses to connect to the server. Moved from local to global.

justMatched flag to indicate if a player was just matched into a game

isChallenger flag to indicare if the player was the one who used “PLAY” to start a match

**FUNCTIONS**

**parseInput()** function used to handle user input

**parseMessage()** function used to handle responses from the server

**CODE**

**parseInput()**

Handles the commands, ‘help’, ‘login’, ‘place’, ‘game’, ‘who’, and ‘play’ commands the same way as clientp1.py’s main function did, and terminates the function upon handling the command. However, it handles game overs differently.

**Receiving “WON”, “LOST”, “TIED”, or “LEFT” messages from server**

Returns a message stating that the player has been sent back to the lobby and terminates the function.

**parseMessage()**

**Handling others attempting to play against the player**

If the player is not a challenger and receives a “MATCHED” message from the server, the client sets inGame to True and justMatched to True, and sends an “OK” message to the server. If the player receives an “OKAY” message instead while justMatched is True, then it is the return message from the previously sent “OK” message, and justMatched is set to False.

**A game has started**

If the player is the challenger, he/she has the first turn, and is notified. Otherwise, he/she is notified to wait for the other player to end their turn. The player is notified of the other player’s player name and the board is displayed.

**main()**

**Setting handlers**

Instead of directly handling commands in the main function, the main function sets up selectors to notify clientSocket when reading/writing is to be done.

“clientSelectors.register(clientSocket, selectors.EVENT\_READ, parseMessage)” sets it up so that any messages passed to clientSocket is handled by parseMessage().

“clientSelectors.register(sys.stdin, selectors.EVENT\_READ, parseInput)” sets it up so that any messages passed to stdin (the command line) is handled by parseInput().

**Handling commands and responses**

The main method has loop that runs indefinitely that triggers the selectors to handle commands and responses from the server when they are passed to the client.

**======================TESTING DOCUMENTATION**======================

**CLIENT-SERVER CASES (PART ONE)**

**NOTE: For all subsequent images, the server is on the left and the two clients are on the right.**

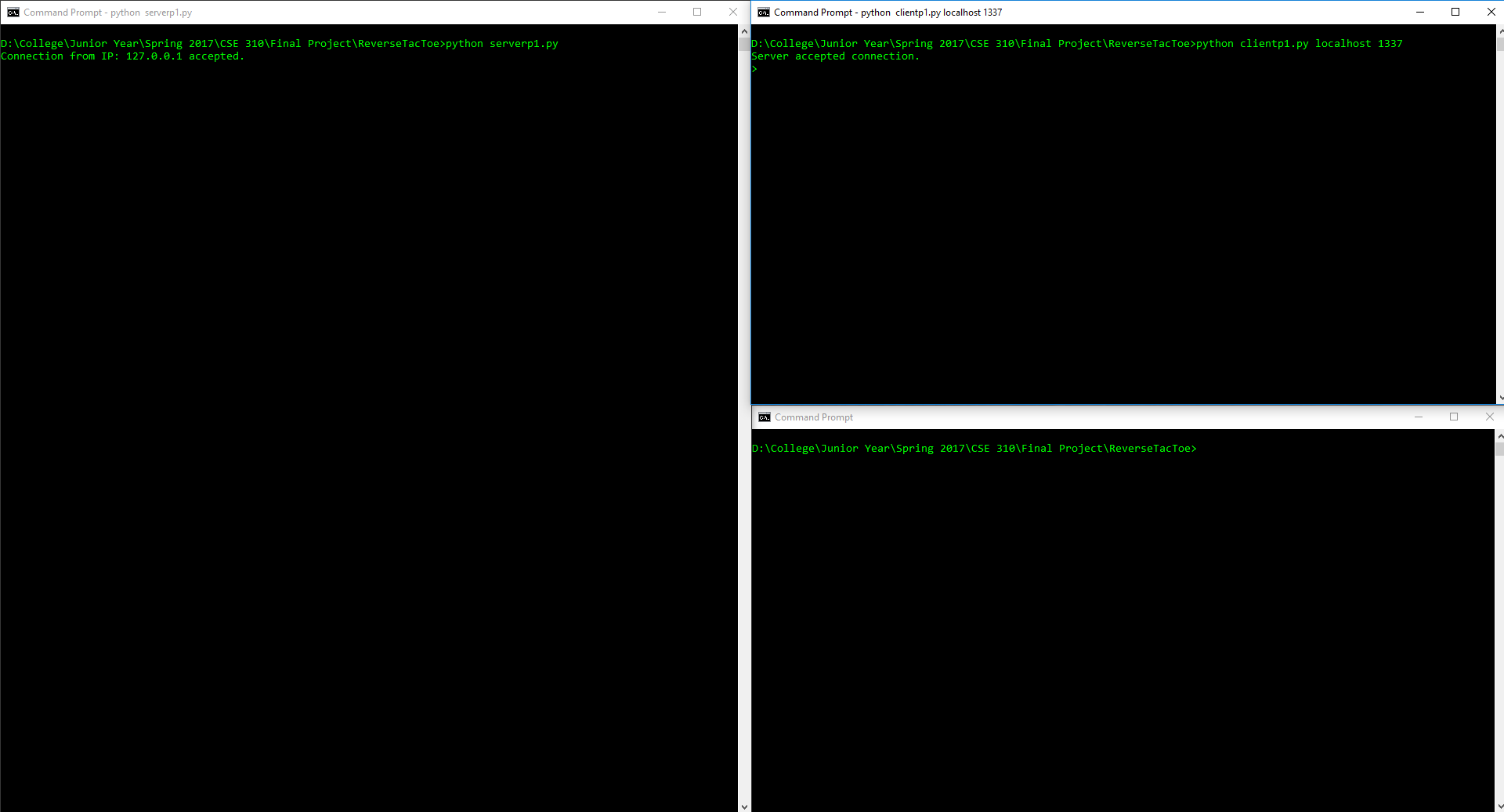


Figure 1: Server started and one client successfully connected

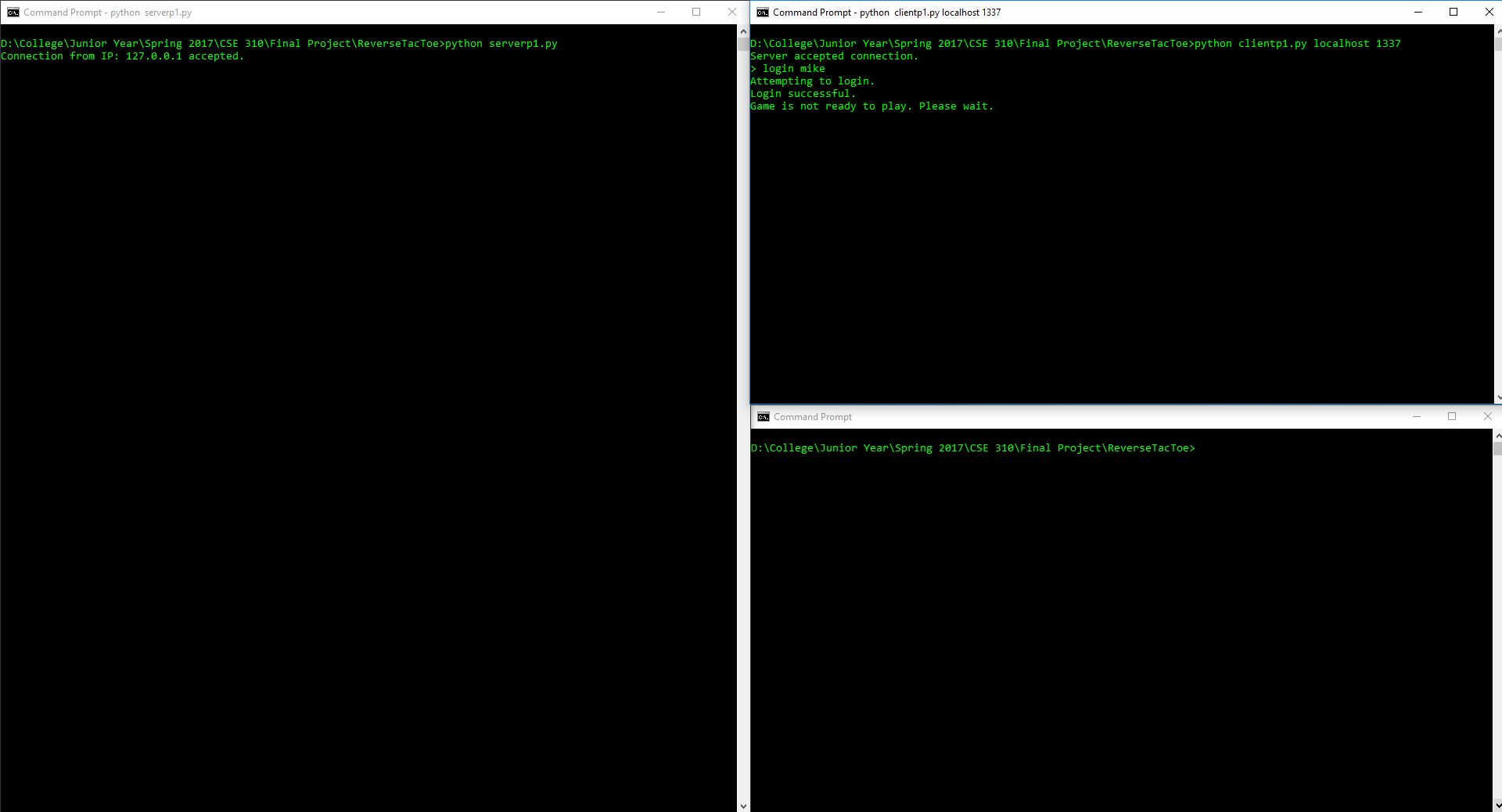


Figure 2: Connected client successfully logs in with userId: mike

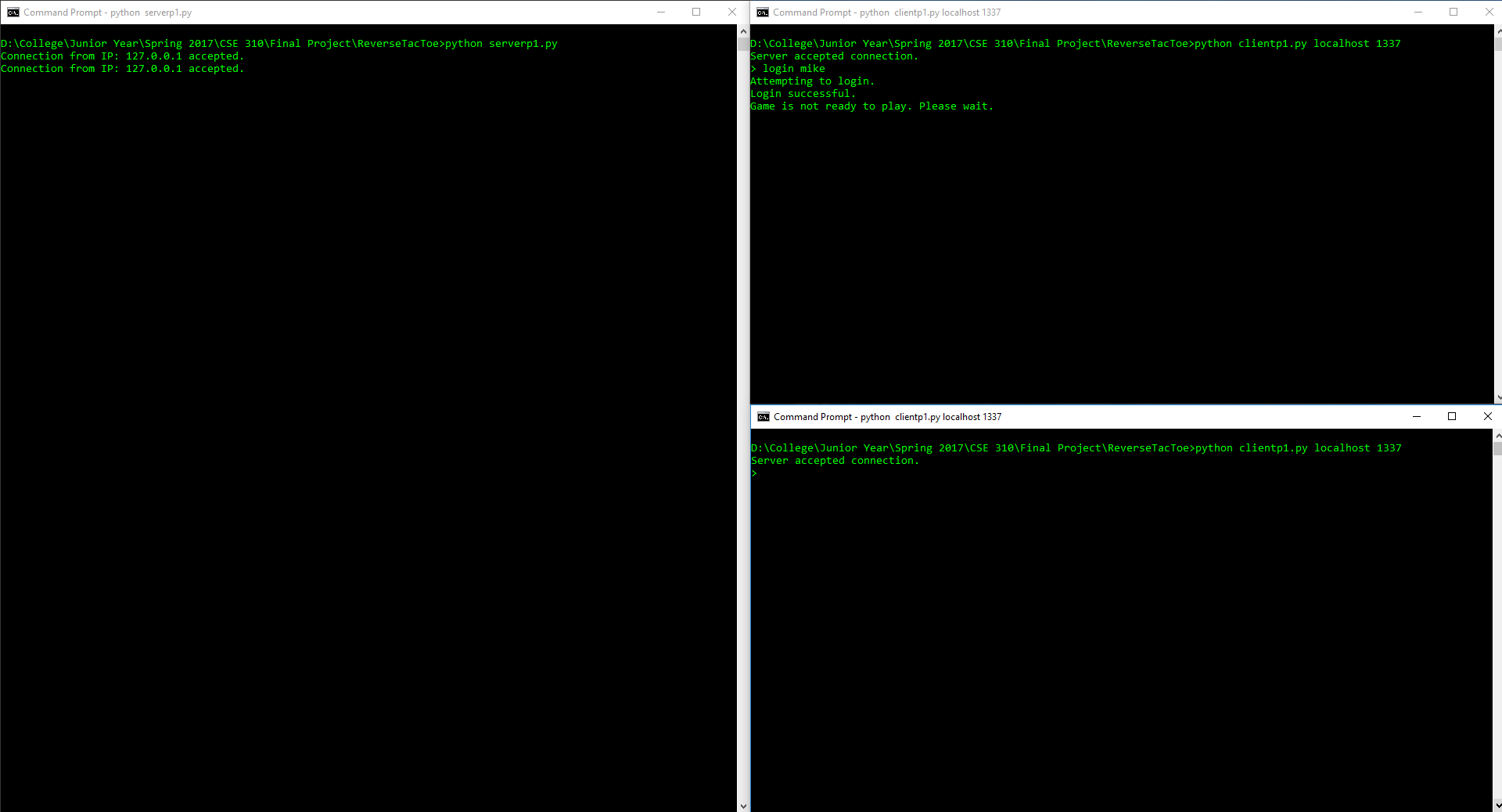


Figure 3: Second client successfully connected

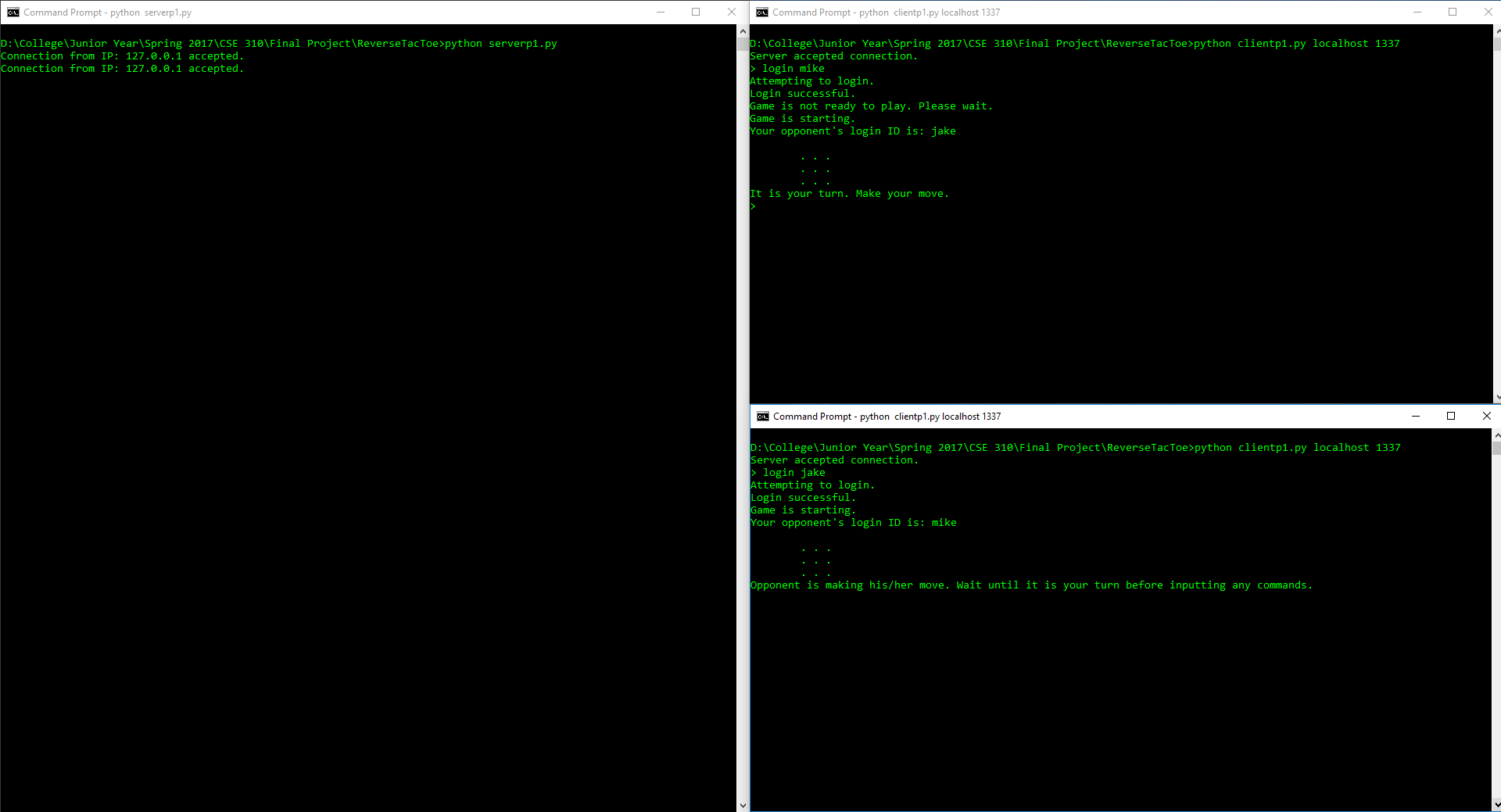


Figure 4: Second client successfully logs in – players are auto-matched and game is started

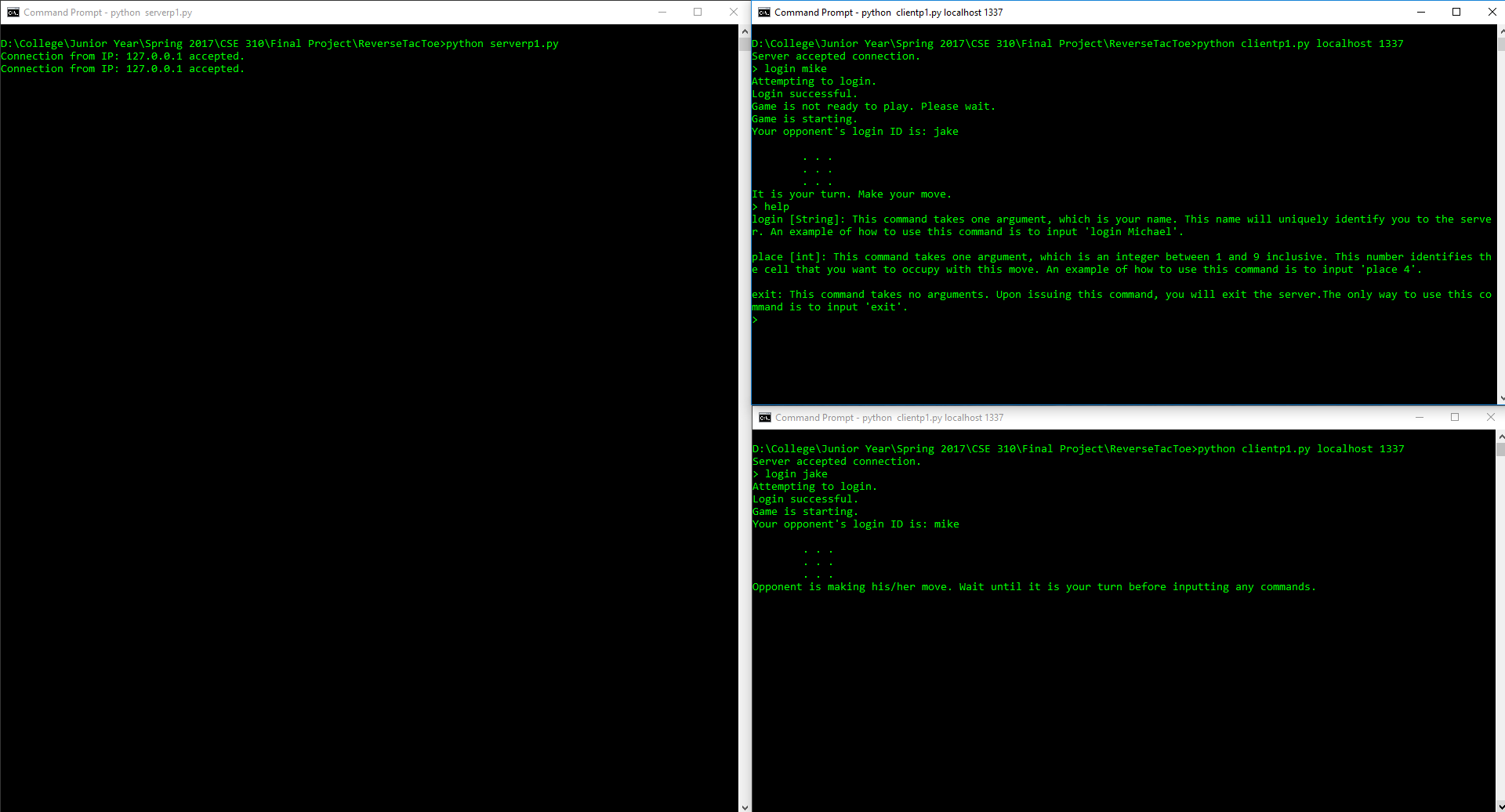


Figure 5: Help command is used mid-game

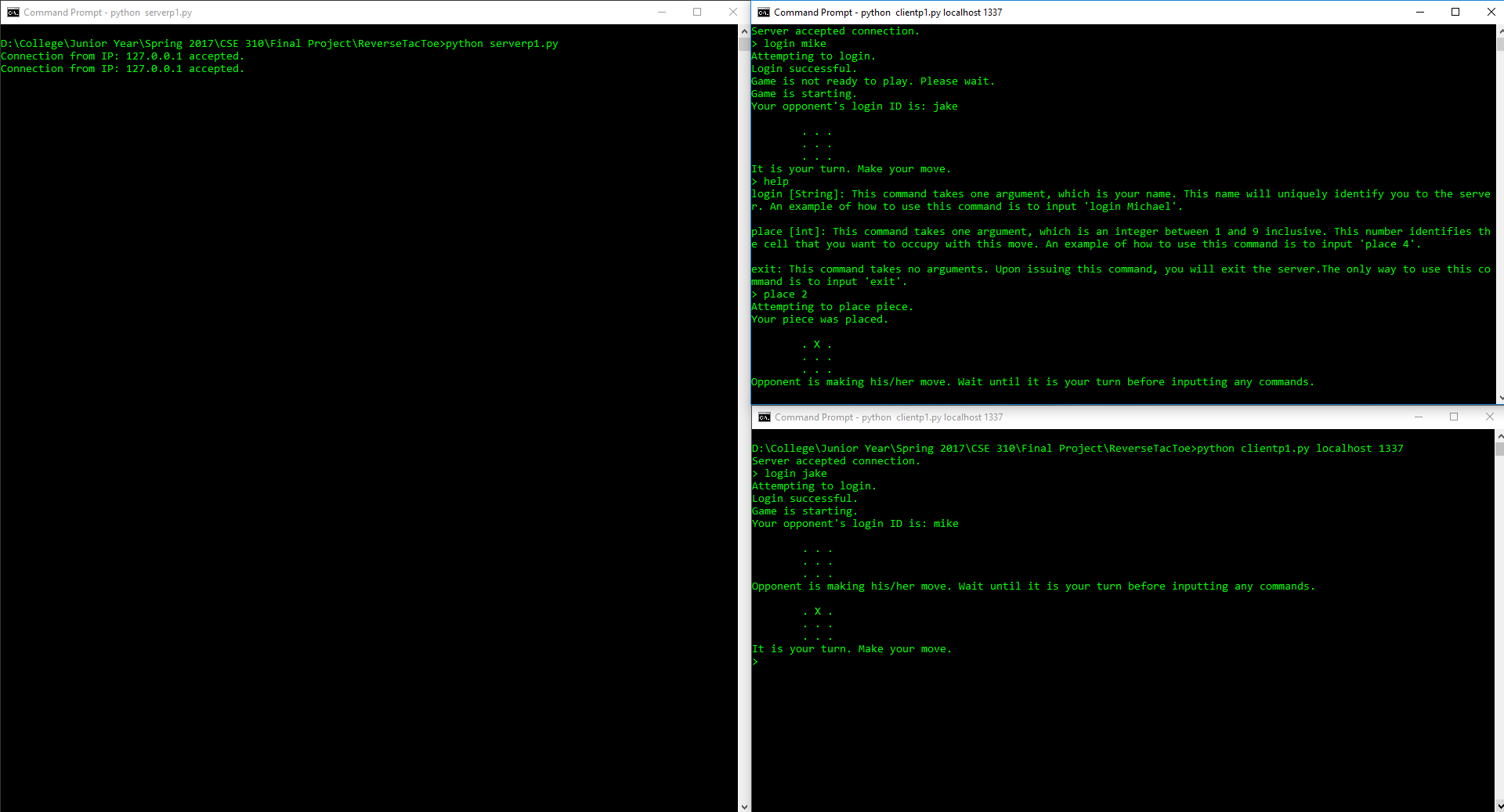


Figure 6: First client makes a move and the board is updated to reflect it

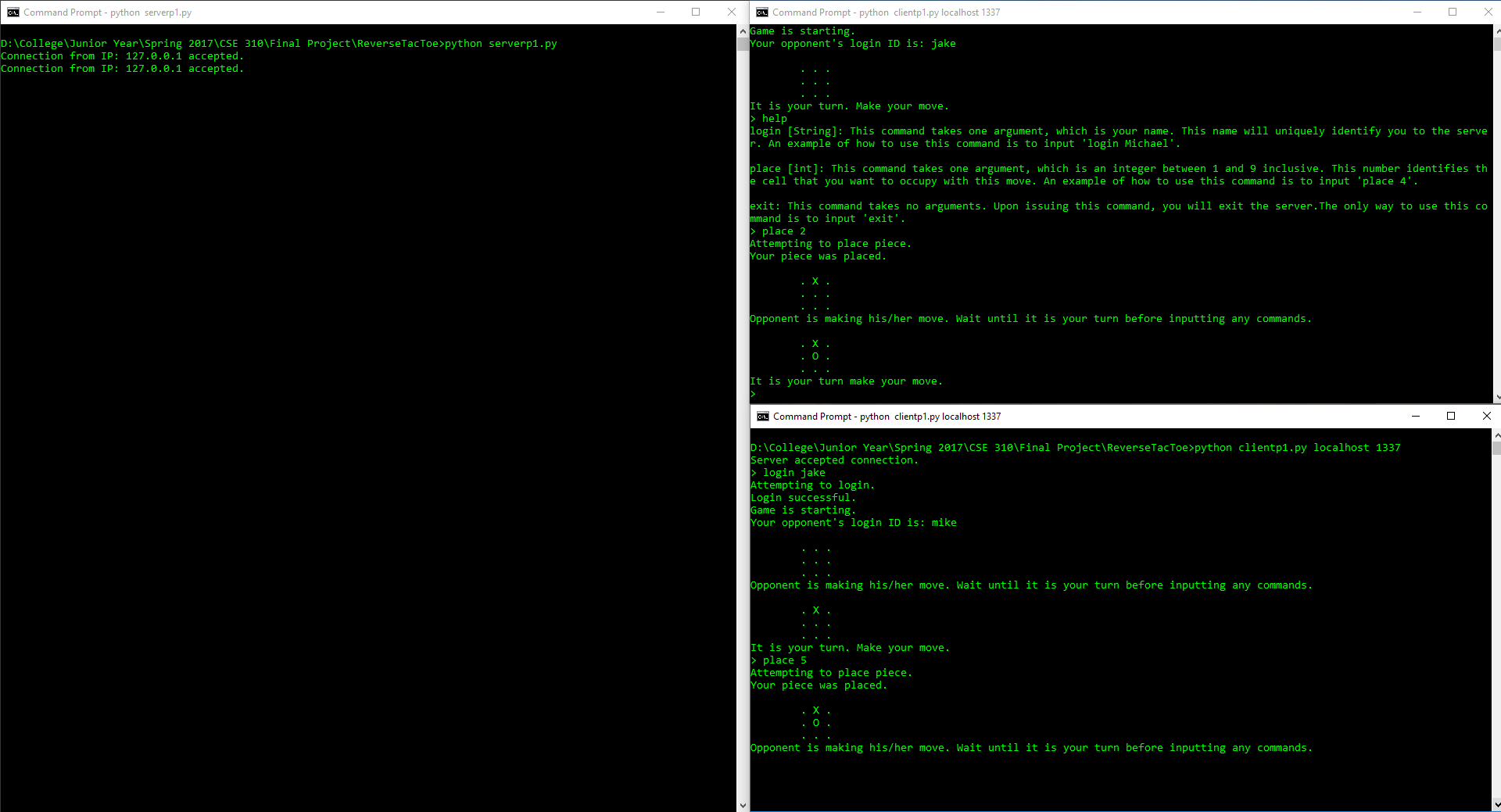


Figure 7: Second client makes a move and the board is updated to reflect it

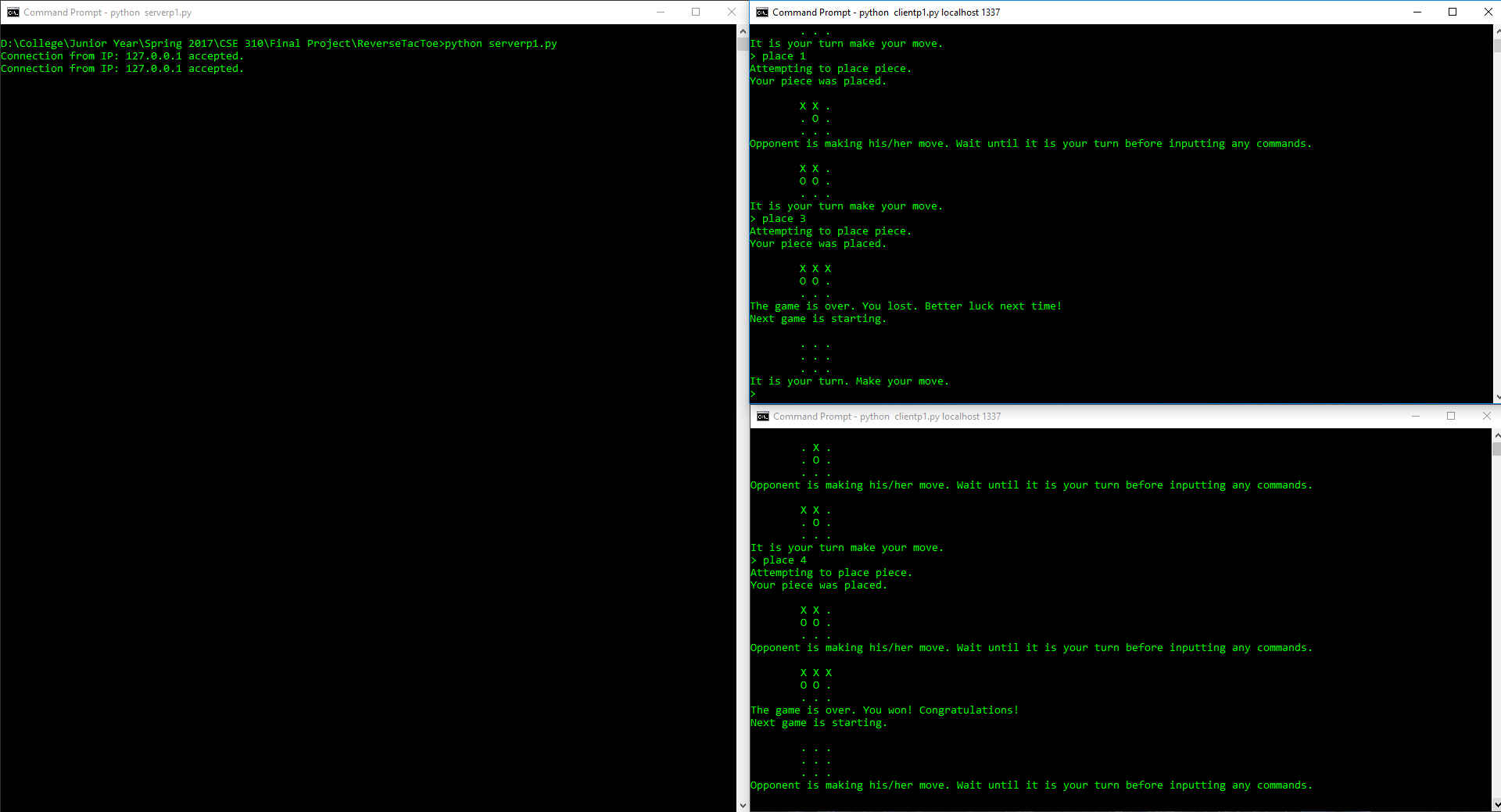


Figure 8: Game is finished – first client loses. Game then restarts

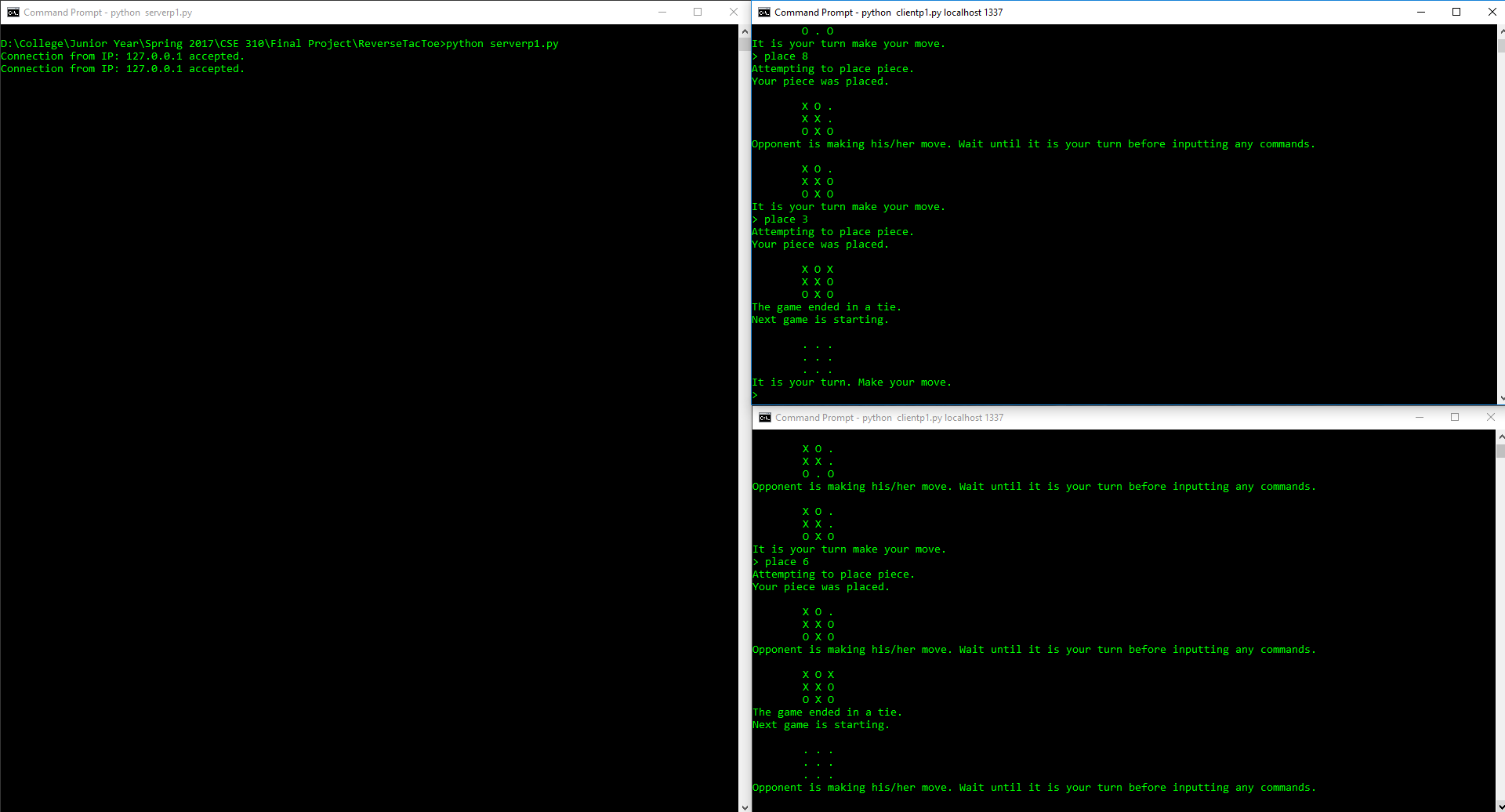


Figure 9: Game is finished – clients tie. Game is then restarted

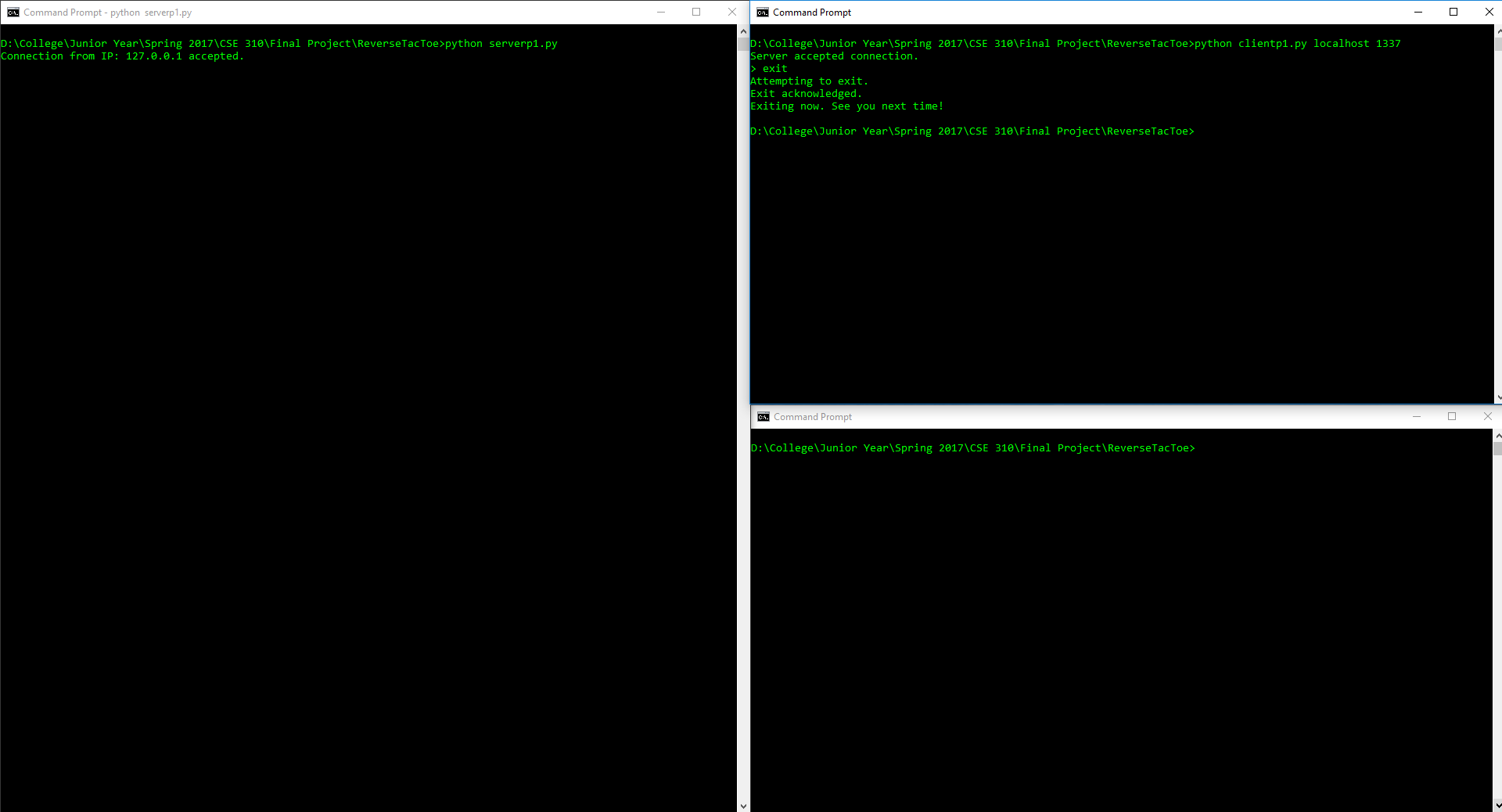


Figure 10: First client immediately exits after connecting

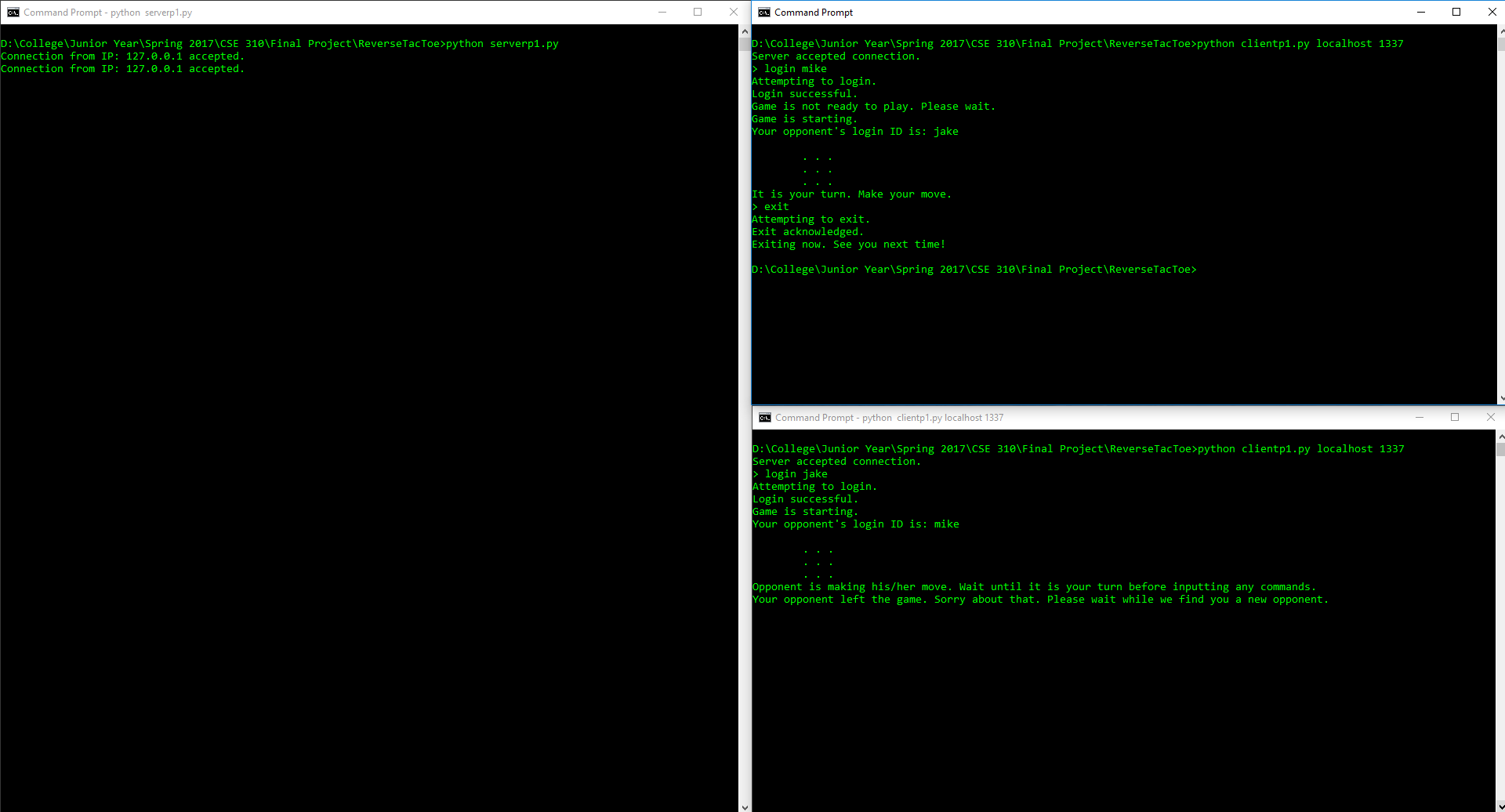


Figure 11: First client exits mid-game – second client waits for new opponent

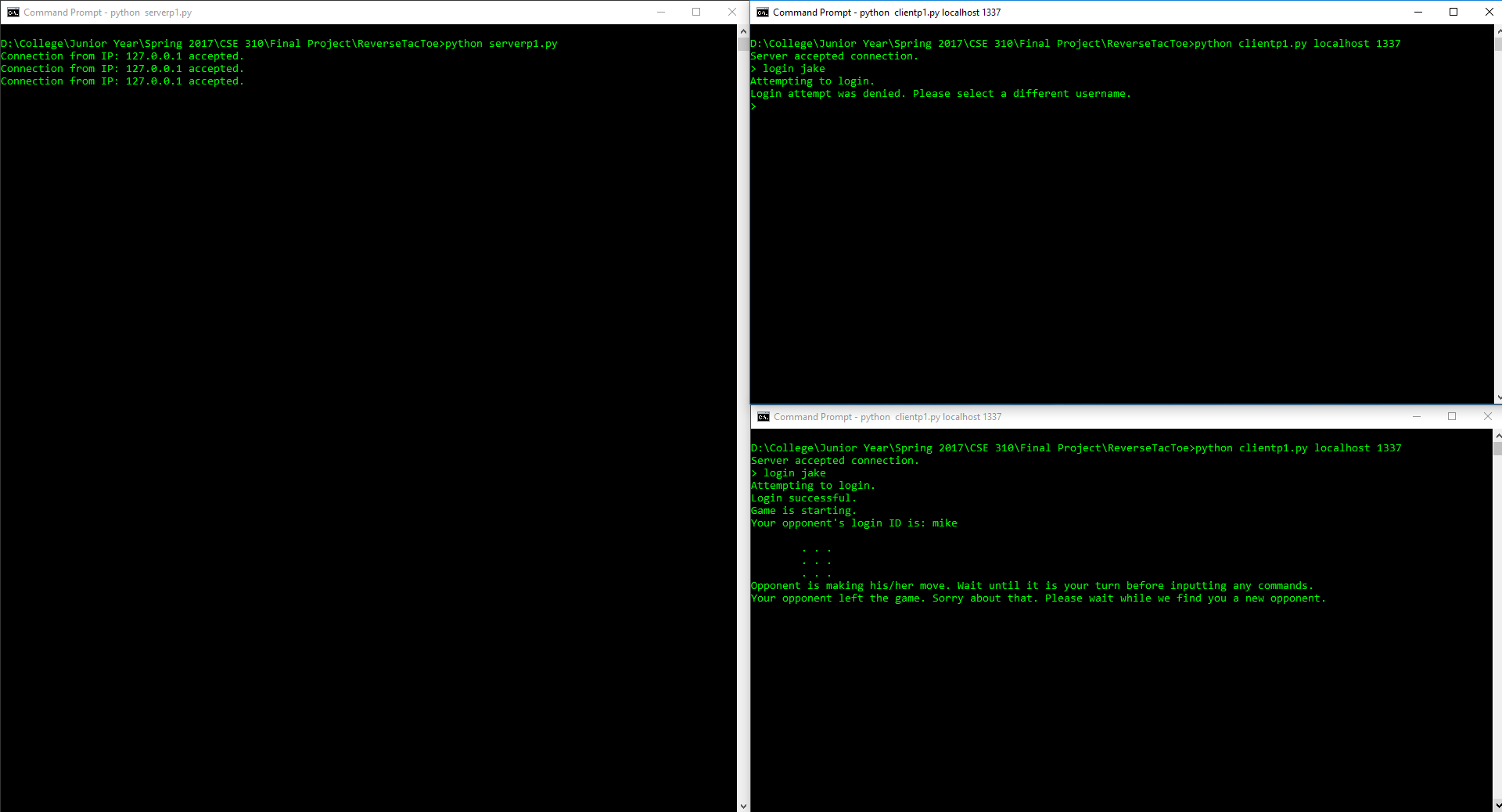


Figure 12: First client unsuccessfully tries to login with a username already in use

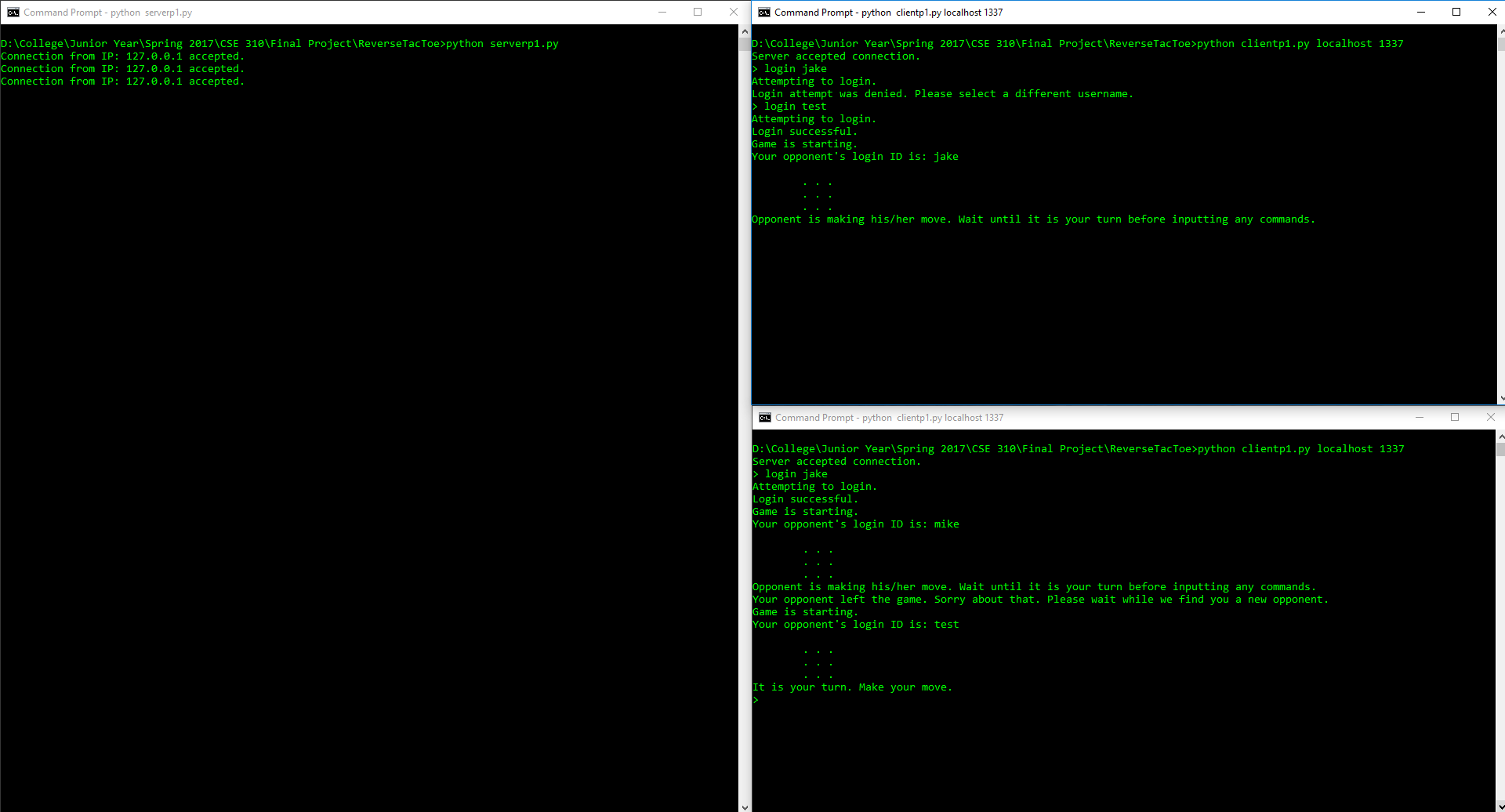


Figure 13: First client logs in and is auto-matched with existing player (after initial exit)

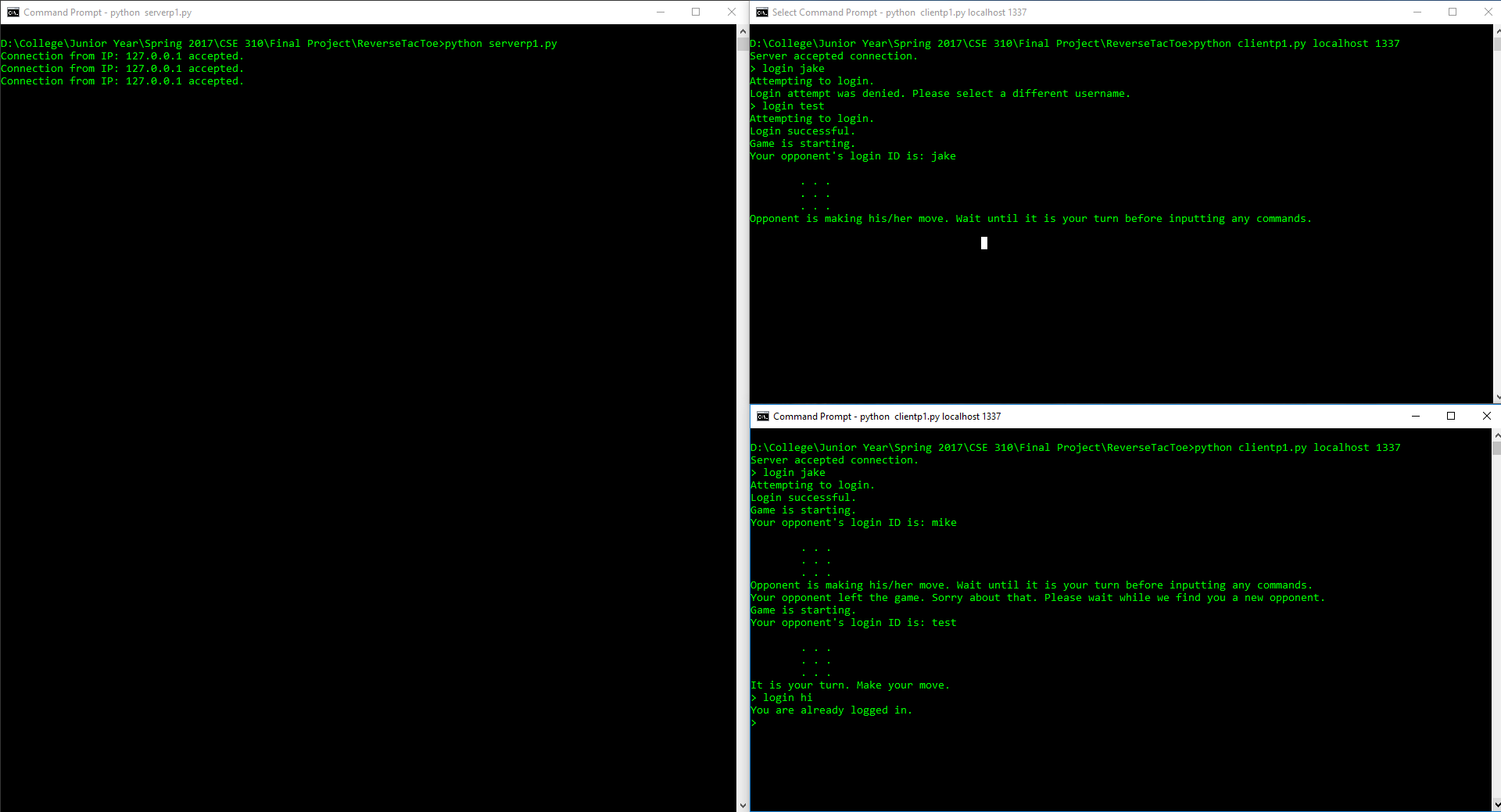


Figure 14: Second client unsuccessfully tries to login while already logged in