

TIC TAC TOE GAME



**Submitted by**

| **Name 01: Hans Upadhyay** | **Reg. No.: 2141021002** |
| --- | --- |
| **Name 02: Sachidananda Tripathy** | **Reg. No.: 2141020055** |
| **Name 03: Krishna Vamsi** | **Reg. No.: 2141022009** |
| **Name 04: Rahul Kumar Yadav** | **Reg. No.: 2141025004** |
| **Name 05: Vikramaditya** | **Reg. No.: 2141022006** |

**B. Tech. CSIT 5th Semester (Section E)**

# Declaration

We, the undersigned students of B. Tech. of **CSIT** Department hereby declare that we own the full responsibility for the information, results etc. provided in this PROJECT titled “**TIC TAC TOE GAME**” submitted to **Siksha ‘O’ Anusandhan (Deemed to be University), Bhubaneswar** for the partial fulfillment of the subject **Computer Networking (CSE 3034)**. We have taken care in all respect to honor the intellectual property right and have acknowledged the contribution of others for using them in academic purpose and further declare that in case of any violation of intellectual property right or copyright we, as the candidate(s), will be fully responsible for the same.

| **Hans Upadhyay**  **Registration No.: 2141021002** | **Sachidananda Tripathy**  **Registration No.: 2141020055** |
| --- | --- |
| **Krishna Vamsi**  **Registration No.: 2141022009** | **Rahul Kumar Yadav**  **Registration No.: 2141025004** |
| **Vikramaditya**  **Registration No.: 2141022006** | |

**DATE: 13/01/2024**

**PLACE: Bhubaneswar, Odisha**

# Abstract

This project presents a comprehensive console-based client-server application for a two-player Tic Tac Toe game implemented in Java. The architecture follows a client-server model, facilitating robust networked interactions through socket programming. The server, capable of handling multiple concurrent connections, utilizes multithreading to ensure optimal performance. This design enables players to engage in a shared gaming experience, where each client communicates seamlessly with the server.

Critical to the project's success is the implementation of game state synchronization, guaranteeing that all clients are aware of the current state of the game. This synchronization facilitates smooth gameplay interactions, allowing players to make moves and receive real-time updates on the evolving game board. The console interface provides an intuitive platform for players to input moves and actively participate in the Tic Tac Toe experience.

The project serves as an educational resource, offering insights into key networking concepts such as socket programming, client-server architecture, and multithreading. By focusing on a widely recognized and simple game like Tic Tac Toe, this application serves as an accessible entry point for understanding more complex aspects of networked application development. It stands as a practical illustration of how to create interactive, synchronized, and concurrent systems using Java, laying a foundation for future exploration into sophisticated networked gaming and communication applications.

# Contents

| **Serial**  **No.** | **Chapter No.** | **Title of the Chapter** | **Page No.** |
| --- | --- | --- | --- |
| 1. | 1 | Introduction |  |
| 2. | 2 | Problem Statement |  |
| 3. | 3 | Methodology |  |
| 4. | 4 | Implementation |  |
| 5. | 5 | Results and interpretation |  |
| 6. | 6 | Conclusion |  |
| 7. |  | References |  |
|  |  |  |  |

**1. Introduction**

This project introduces a robust console-based client-server application designed to facilitate a two-player Tic Tac Toe game, employing advanced Java programming concepts. The client-server architecture enables seamless communication between multiple players, fostering an interactive gaming experience. Leveraging socket programming, the application ensures efficient data exchange, allowing players to make moves and receive real-time updates on the game state.

A fundamental feature of the project is the incorporation of server-side multithreading, a critical aspect in managing concurrent connections effectively. This design choice optimizes the responsiveness of the server, allowing it to handle multiple players concurrently without sacrificing performance. The game state synchronization mechanism guarantees that all clients are consistently informed about the ongoing game, providing a shared environment for players.

The console interface serves as the interactive platform, allowing players to input moves and actively engage in the Tic Tac Toe match. The simplicity of the game enhances its educational value, making it an ideal starting point for understanding essential networking concepts, including socket communication, client-server models, and multithreading.

This project not only serves as a practical application of theoretical networking concepts but also acts as a stepping stone for developers aspiring to delve deeper into the intricacies of networked application development. Through its focus on simplicity and clarity, this Tic Tac Toe game lays the groundwork for future exploration into more complex and sophisticated networked applications in the realm of gaming and communication.

# 2. Problem Statement

**I. Problem Statement: Console-Based Tic Tac Toe Game with Data Persistence:**

The objective of this project is to develop a console-based client-server application for a two-player Tic Tac Toe game in Java. The challenge involves implementing a user-friendly interface where players can input their moves through the console, facilitating interactive gameplay. The system should seamlessly identify and process user inputs, updating the game state accordingly. The goal is to create a synchronized experience where both players receive real-time feedback on the evolving game board.

Furthermore, to enhance the user experience, the project aims to incorporate data persistence. The outcome of each game session should be reflected in the form of file content or stored in a database accessible via the console. This feature ensures that players can review past game results, adding a layer of engagement and functionality beyond the immediate gaming session.

**II. Constraints:**

**Console Interface Limitations:** The interface must be designed to accommodate user inputs effectively within a console environment, considering the limitations of text-based interactions.

**Real-time Synchronization:** Achieving real-time synchronization between the server and clients while maintaining responsiveness poses a challenge, especially in a multithreaded environment.

**Data Persistence Overhead:** Incorporating data persistence requires careful consideration of file handling or database management, introducing complexity to the project.

**User Input Validation:** The system should robustly validate user inputs to prevent invalid moves and maintain the integrity of the game state.

Addressing these challenges will contribute to the successful implementation of a console-based Tic Tac Toe game with seamless user interaction and data persistence capabilities.

# 3. Methodology

The algorithmic approach outlines the key steps involved in the console-based Tic Tac Toe game from both the server and client perspectives, emphasizing effective communication, move validation, and game state synchronization. The implementation of the program/pseudocode is mentioned below:

**Server-Side Algorithm:**

1. **Initialize the Game State:**

* Create a data structure to represent the Tic Tac Toe board.
* Initialize the board to an empty state.

1. **Accept Player Connections:**

* Use a ServerSocket to listen for incoming connections.
* Upon connection, spawn a new PlayerHandler thread to manage communication with the connected client.

1. **Handle Player Moves:**

* Receive player moves from clients through the socket.
* Validate the moves to ensure they are within the bounds of the board and represent an unoccupied space.
* Update the game state accordingly.

1. **Check for Win or Draw:**

* After each move, check if the current player has won or if the game is a draw.
* Communicate the result to the players.

1. **Update Clients:**

* Send the updated game state to all connected clients after each move or game outcome.
* Ensure proper synchronization to maintain a shared game experience.

**Client-Side Algorithm:**

1. **Connect to Server:**

* Use a Socket to connect to the server.
* Initialize the Game Interface:
* Display the initial state of the Tic Tac Toe board in the console.

1. **Receive and Display Updates:**

* Continuously listen for updates from the server.
* Update the console interface based on the received game state.

1. **Accept User Input:**

* Prompt the user to input their move through the console.
* Validate the input to ensure it is a valid move.

1. **Send Moves to Server:**

* Send the validated move to the server through the socket.

1. **Receive Game Outcome:**

* Listen for the final outcome of the game (win, draw, or ongoing).
* Display the result in the console.

1. **Close Connection:**

* Close the socket connection when the game concludes.

**ClientHandler Algorithm:**

1. **Initialize Connections:**

* Accept two client sockets (clientSocket1 and clientSocket2) from the server.
* Create ObjectInputStream and ObjectOutputStream for each client socket to handle input and output streams.

1. **Initialize Game State:**

* Initialize the Tic Tac Toe board as a 3x3 array of characters.
* Set the initial player (currentPlayer) to 'X'.

1. **Welcome Messages:**

* Send welcome messages to both clients, indicating their assigned player ('X' or 'O').

1. **Game Loop:**

* Enter an infinite loop to manage the ongoing game.
* Continuously send the current state of the board to both clients for display.

1. **Player Move (X):**

* If it's player 'X's turn:
* Send a prompt to player 'X' to enter a move in the format "row column" (e.g., "11" for the top-left cell).
* Receive the move as a string from player 'X'.
* Convert the move to row and column indices and validate if it's a valid move.
* If valid, update the board, check for a winner, switch player if needed, and continue the loop.
* If invalid, send an error message to player 'X'.

1. **Player Move (O):**

* If it's player 'O's turn:
* Send a prompt to player 'O' to enter a move.
* Receive the move as a string from player 'O'.
* Convert the move to row and column indices and validate if it's a valid move.
* If valid, update the board, check for a winner, switch player if needed, and continue the loop.
* If invalid, send an error message to player 'O'.

1. **Game Outcome:**

* Check for a winner or a tie after each move.
* If a winner is found, send win/lose messages to the respective players.
* If a tie is detected, send tie messages to both players.

1. **Close Connections:**

* Close both client sockets when the game concludes.

# 4. Implementation

**1. Server.java**

import java.io.\*;

import java.net.ServerSocket;

import java.net.Socket;

public class Server {

public static void main(String[] args) {

try {

ServerSocket serverSocket = new ServerSocket(6370);

System.out.println("Server started. Waiting for clients...");

while (true) {

Socket clientSocket1 = serverSocket.accept();

System.out.println("Client connected: " + clientSocket1);

Socket clientSocket2 = serverSocket.accept();

System.out.println("Client connected: " + clientSocket2);

ClientHandler clientHandler = new ClientHandler(clientSocket1, clientSocket2);

Thread thread = new Thread(clientHandler);

thread.start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**2. Client.java**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class Client {

public static void main(String[] args) {

try {

Scanner sc = new Scanner(System.in);

int port = 6370;

Socket socket = null;

System.out.println("trying port " + port);

socket = new Socket("192.168.24.17", port);

System.out.println("Connected to the server.");

ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream());

ObjectOutputStream outputStream = new

ObjectOutputStream(socket.getOutputStream());

while (true) {

String message = (String) inputStream.readObject();

if (message.contains("Board"))

System.out.print("\033[H\033[2J");

System.out.println(message);

if (message.contains("Enter row")) {

String move = sc.nextLine();

outputStream.writeObject(move);

outputStream.flush();

}

if (message.contains("wins") || message.contains("tie") || message.contains("lose")) {

break;

}

}

sc.close();

socket.close();

} catch (Exception e) {

//e.printStackTrace();

}

}

}

**3. ClientHandler.java**

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

public class ClientHandler implements Runnable {

private Socket clientSocket1;

private Socket clientSocket2;

private ObjectInputStream inputStream1;

private ObjectOutputStream outputStream1;

private ObjectInputStream inputStream2;

private ObjectOutputStream outputStream2;

private char[][] board;

private char currentPlayer;

public ClientHandler(Socket clientSocket1, Socket clientSocket2) {

this.clientSocket1 = clientSocket1;

this.clientSocket2 = clientSocket2;

try {

outputStream1 = new ObjectOutputStream(clientSocket1.getOutputStream());

inputStream1 = new ObjectInputStream(clientSocket1.getInputStream());

outputStream2 = new ObjectOutputStream(clientSocket2.getOutputStream());

inputStream2 = new ObjectInputStream(clientSocket2.getInputStream());

board = new char[3][3];

currentPlayer = 'X';

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

try {

sendMessage1("Welcome to Tic Tac Toe! You are player " + currentPlayer);

sendMessage2("Welcome to Tic Tac Toe! You are player " + ((currentPlayer == 'X') ? 'O' : 'X'));

while (true) {

sendMessage1("Current Board:\n" + displayBoard());

sendMessage2("Current Board:\n" + displayBoard());

if (currentPlayer == 'X') {

sendMessage1("Enter row (1-3) and column (1-3) e.g. 11 : \nYour move (" + currentPlayer + ") : ");

sendMessage2("Move for X");

String move = (String) inputStream1.readObject();

int row = (move.charAt(0) - 49);

int col = (move.charAt(1) - 49);

if (isValidMove(row, col)) {

makeMove(row, col);

if (isWinner()) {

printWin(currentPlayer);

break;

} else if (isBoardFull()) {

printTie();

break;

} else {

switchPlayer();

}

} else {

sendMessage1("Invalid move. Try again.");

}

} else {

sendMessage2("Enter row (1-3) and column (1-3) e.g. 11 : \nYour move (" + currentPlayer + ") : ");

sendMessage1("Move for 0");

String move = (String) inputStream2.readObject();

int row = (move.charAt(0) - 49);

int col = (move.charAt(1) - 49);

if (isValidMove(row, col)) {

makeMove(row, col);

if (isWinner()) {

printWin(currentPlayer);

break;

} else if (isBoardFull()) {

printTie();

break;

} else {

switchPlayer();

}

} else {

sendMessage2("Invalid move. Try again.");

}

}

}

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

} finally {

try {

clientSocket1.close();

clientSocket2.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

private void printTie() {

try {

sendMessage1(displayBoard() + "\nThe game is a tie!");

sendMessage2(displayBoard() + "\nThe game is a tie!");

} catch (Exception ex) {

ex.printStackTrace();

}

}

private void printWin(char ch) {

try {

sendMessage1("Current Board:\n" + displayBoard());

sendMessage2("Current Board:\n" + displayBoard());

if (ch == 'X') {

sendMessage1("\nYou win!");

sendMessage2("\nYou lose");

} else {

sendMessage2("\nYou win!");

sendMessage1("\nYou lose");

}

} catch (Exception e) {

//e.printStackTrace();

}

}

private boolean isValidMove(int row, int col) {

return row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == '\0';

}

private void makeMove(int row, int col) {

board[row][col] = currentPlayer;

}

private void switchPlayer() {

currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';

}

private boolean isWinner() {

for (int i = 0; i < 3; i++) {

if (board[i][0] == currentPlayer && board[i][1] == currentPlayer && board[i][2] == currentPlayer) {

return true; // Check rows

}

if (board[0][i] == currentPlayer && board[1][i] == currentPlayer && board[2][i] == currentPlayer) {

return true; // Check columns

}

}

return (board[0][0] == currentPlayer && board[1][1] == currentPlayer && board[2][2] == currentPlayer)

|| (board[0][2] == currentPlayer && board[1][1] == currentPlayer && board[2][0] == currentPlayer);

}

private boolean isBoardFull() {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == '\0') {

return false;

}

}

}

return true; // Board is full, and no one has won

}

private String displayBoard() {

StringBuilder display = new StringBuilder();

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

display.append(board[i][j] == '\0' ? " - " : " " + board[i][j] + " ");

if (j < 2) {

display.append("|");

}

}

display.append("\n");

if (i < 2) {

display.append("-----------\n");

}

}

return display.toString();

}

private void sendMessage1(String message) throws IOException {

outputStream1.writeObject(message);

outputStream1.flush();

}

private void sendMessage2(String message) throws IOException {

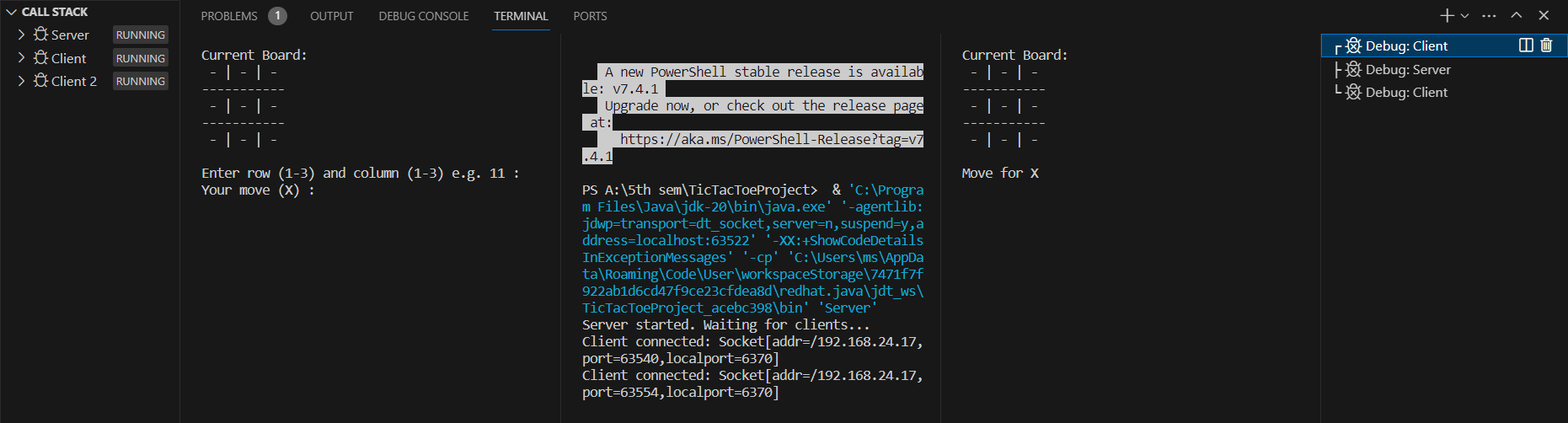
outputStream2.writeObject(message);

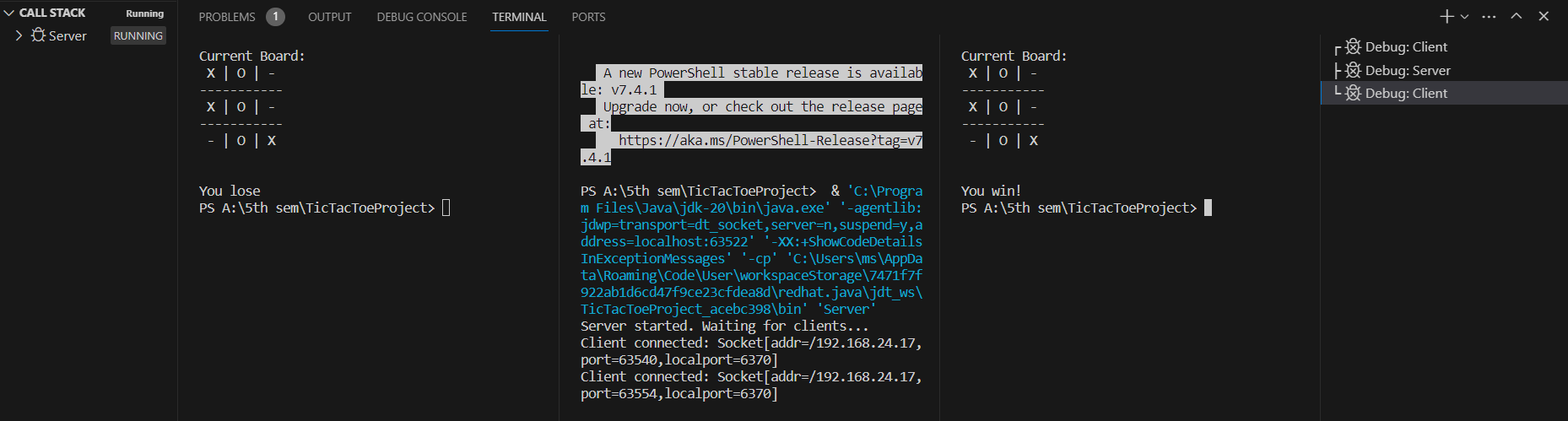
outputStream2.flush();

}

}

# 5. Results & Interpretation





# 6. Conclusion

In conclusion, the development of a console-based client-server application for a two-player Tic Tac Toe game in Java has yielded a robust and educational project. By adhering to the principles of socket programming, client-server architecture, and multithreading, this undertaking demonstrates a practical application of fundamental networking concepts.

The project successfully addresses the challenge of providing an interactive gaming experience through a user-friendly console interface, allowing players to make moves and receive real-time updates on the evolving game state.

The incorporation of server-side multithreading enhances the system's efficiency, enabling it to handle concurrent player connections seamlessly. Additionally, the emphasis on game state synchronization ensures a shared environment, fostering a synchronized and engaging gameplay experience for all participants. The project's educational value extends beyond mere gameplay, serving as a foundational exploration into the intricacies of networked application development.

Moreover, the inclusion of data persistence enriches the project's functionality, reflecting a commitment to enhancing user experience by storing game results in a file or database. This feature provides players with the opportunity to review past game outcomes, adding depth and continuity to the gaming experience.

By successfully navigating through the challenges of real-time synchronization, user input validation, and data persistence integration, this project serves as a stepping stone for developers aiming to delve into more complex networked applications. In essence, the console-based **Tic Tac Toe game** exemplifies an accessible yet comprehensive demonstration of Java programming for networked environments, laying a solid foundation for future endeavors in the dynamic field of networked application development.

**References**

(as per the IEEE recommendations)

[1] Computer Networks, Andrew S. Tannenbaum, Pearson India.

[2] Java Network Programming by Harold, O’Reilly (Shroff Publishers).

[3]

[4]