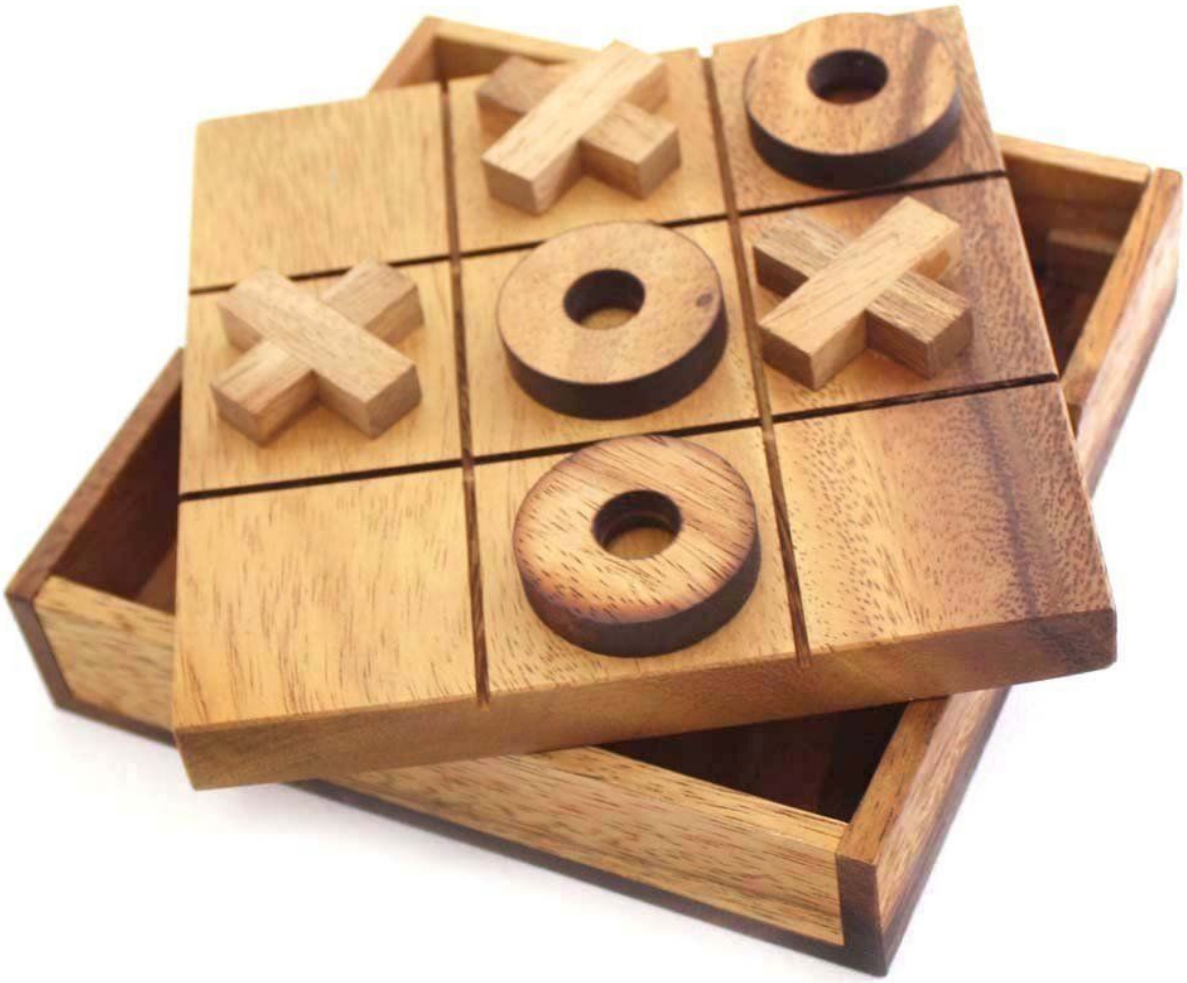
Tic Tac Toe



By: Ravi Bhankharia, Milap Shah, Priyesh Shah

Table of Contents

Introduction/Motivation

For our final project, we decided to implement a multi-player Tic Tac Toe game network. This consisted of a server running a Tic Tac Toe game which allowed multiple users to connect and play each other. We were inspired to do this project not only because of our passion for Tic Tac Toe but also because of the networking involved. All three of us enjoy playing games and so we wanted our final project to be about a game. We wanted to gain experience in the backend matching logic that most online games deal with. We thought of two popular games which were Tic Tac Toe and Chess. After consulting the professor, we realized that Tic Tac Toe involved more networking which was better suited for this course. We were excited to implement this project since it did not only involve good logic reasoning but a good understanding of networking as well. This project encompassed the topics we learned in class well since it involved a wide range of techniques.

Solution

We specifically implemented the project by creating two major c files for our code. One c file was the T3Client.c file which contained the code for the client. This file would be the client side of our code and write messages to server while reading from it as well. In the beginning we establish a connection with the game server since the client needs to connect to it in order to play the game. Once the connection is set up, depending on whether the user is player one or two, they are asked to input a number corresponding to the Tic Tac Toe board which is displayed. Each section of the board has a number from 1-9, so the user can select where they want to play their piece. Once the user inputs a number, there is an error check to ensure the user selected a number within range. If all is right then the number is sent and the server will check if there is already a game piece on that slot or not. This goes back and forth until there is a winner declared or the board is filled up. The other major c file is the T3Server.c file which has the server code. The server creates a new thread whenever a client joins and if the server realized there is more than 2 threads, it forks to create another process to run the other game. Once the two clients are connected the server displays the Tic Tac Toe board to the players. Player 1 makes their move and the server runs error check, such as if there is already a game piece there, and if the move is validated, the server goes to Player 2. This goes back and forth until the server detects a winner or the amount of turns indicates there is a draw. Most of this communication relied on the read and write system calls, and TCP ensured the correct order of the communication.

Functionality

Server-Client Communication

Each client is relatively simple: it connects to the server located on the provided port and sends/receives messages in a while loop. It begins by reading an initial connection message from the server, and then repeats a read-write-read sequence until termination. The first read normally returns the pre-move board state, then the first write sends the player's move to the server and the second read returns the post-move board state. If the server has detected that the game is over, it sends the corresponding message to each client which is picked up by the first read in the client's loop. Because of this special case, the clients compare the output of the first read with messages similar to "You win", "You lose" and "You draw."

Threading and Multiprocessing

In order to ensure that each player has minimal server latency, each player has its own server-side thread that handles client-server communication. In order to ensure that each set of two players has its own copy of a tic tac toe board, the program forks when it connects to two clients. The parent process sleeps while its child threads plays the game and handles each connection. The child process returns to accepting new connection/spawning new threads.

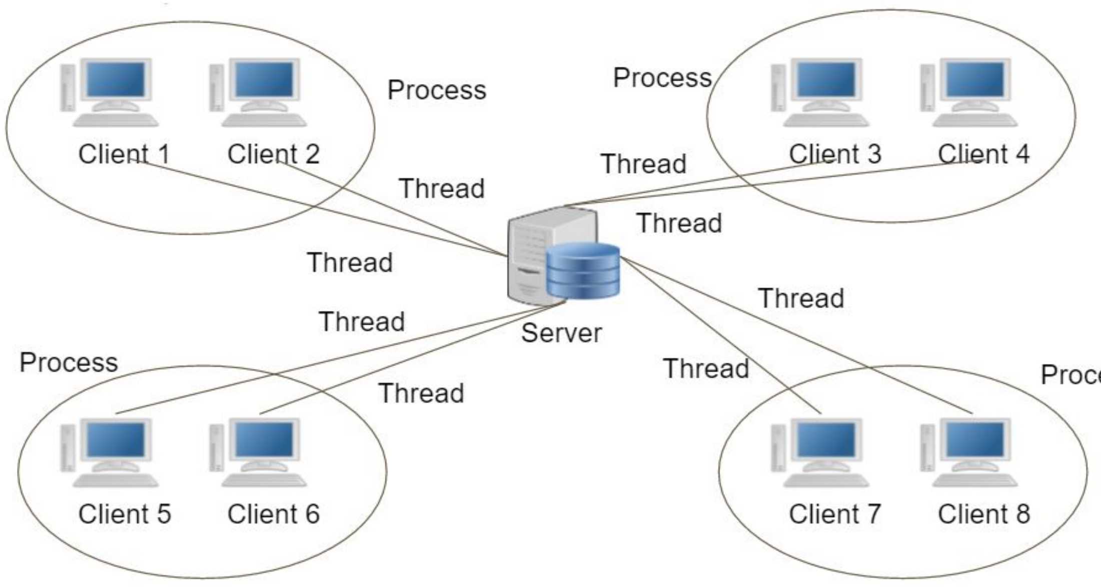
Tic Tac Toe and Synchronization

The server checks the state of the board after each move to see if there are any winning conditions in place or if the game has ended in a draw. The server also ensures that the game is played fairly in terms of only allowing one player to make a move at a time. The board state is also synchronized across the two players for each game, and the server keeps track of all the different board states in the separate processes. This ensures speed and reliability between the server and the client.

Concept Application

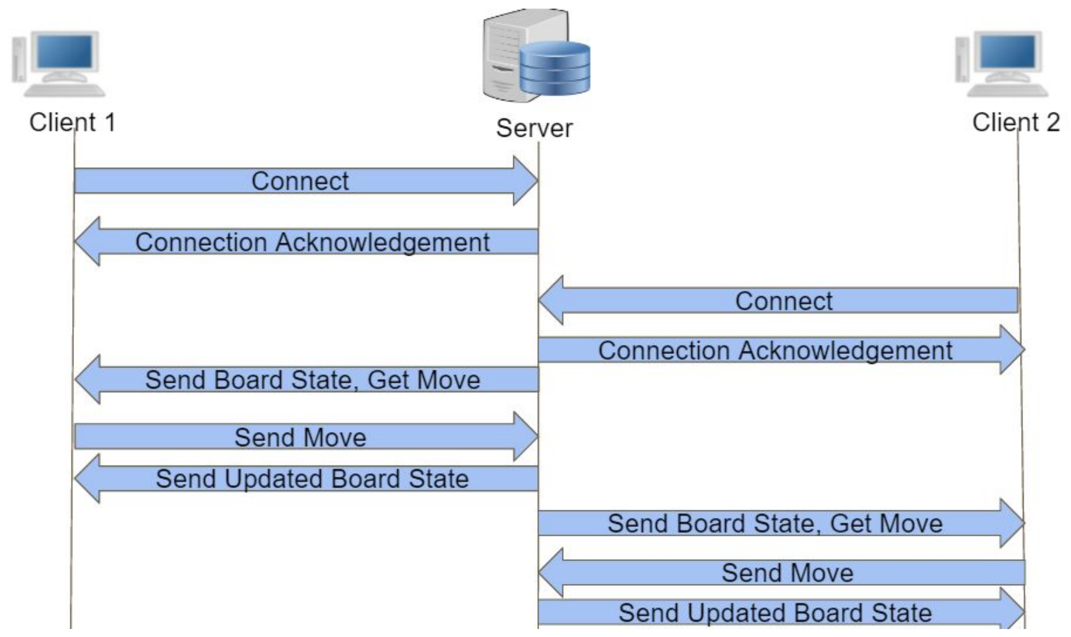
For our solution, we first thought about which network protocol would be best for our game. We examined two main protocols which were UDP and TCP. We observed that UDP had a lower latency and less overhead which would make connections faster. However, TCP ensured a connection and that the data packets would be received in the correct order. We felt that this fact was important in our game and so we went with the clear winner, TCP. We also had to implement IO to communicate with the user and output the game board correctly. We had to constantly prompt the user of their next move but it had to coordinate with the server to let It involved socket networking programming as we implemented our server to communicate with multiple clients and exchange messages efficiently. The bulk of the project revolved around Network Server Design as we had to design the server to handle a load of clients and run simultaneous games. We did this by creating a new thread for each client connection received and creating a new process for a new game. This keeps the network organized and ensures that each client can play the game easily as the next.

Subsystem Breakdown

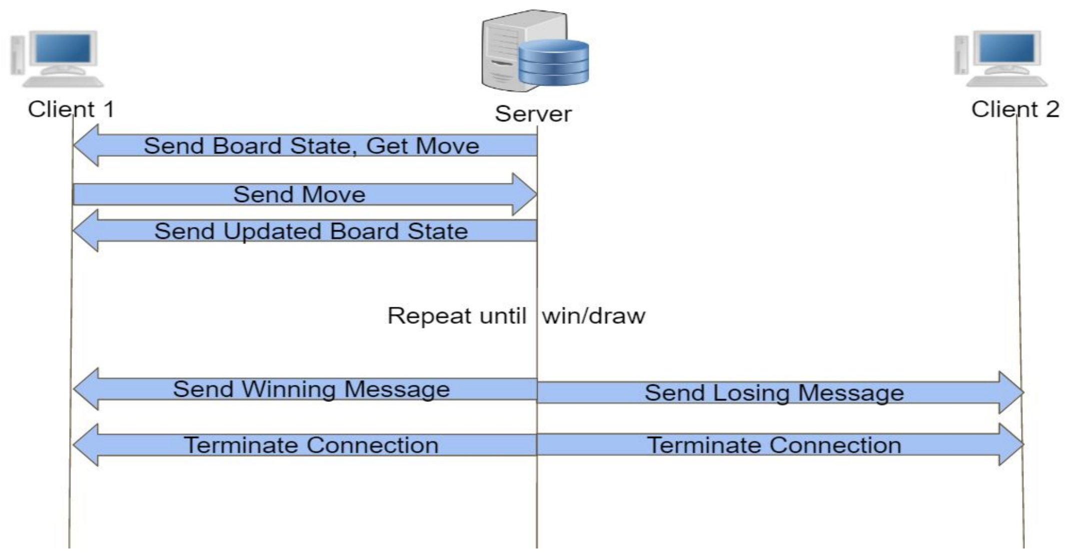


Message Protocol

Initial


Client 1
Server
Client 2
Connect
Connection Acknowledgement
Connect
Connection Acknowledgement
Send Board State, Get Move
Send Move
Send Updated Board State
Send Board State, Get Move
Send Move
Send Updated Board State

End



Contribution Breakdown

Ravi Bhankharia

I implemented the **threading and synchronization** aspects of this project.

Milap Shah

I was tasked with establishing **server-client communication.**

Priyesh Shah

I was in charge of the **Tic Tac Toe and multi-processing** portions of our code and documentation

Appendix A

Located at https://github.com/ravibanks54/TicTacToe

T3Server.c

```
#include "csapp.h"
#include <pthread.h>
#include <unistd.h>

#define DEBUG
int open_clientfd(char *hostname, int port);

int threadCount = 0;    //Per process thread count

char board[3][3] = { // The board
        {'1', '2', '3'}, // Initial values are reference numbers
        {'4', '5', '6'}, // used to select a vacant square for
        {'7', '8', '9'} // a turn.
};
int turn = 0;       // Global Turn Counter
int currentPlayer = -1;
int hodor = 0;      // Signal variable if a player has won

void* handleConnection(void* args);

typedef struct arguments_t {        //Used to create a thread to handle each client connection
    int connfd;
    struct sockaddr_in clientaddr;
    int playerID;                  //Pass the player's ID into the thread
} arguments;

int main(int argc, char **argv)
{
    int listenfd;             // The proxy's listening descriptor
    int port;                 // The port the proxy is listening on
    int clientlen;            // Size in bytes of the client socket address
    struct sockaddr_in clientaddr;
    int connfd;               // The file descriptor for each client
    arguments* args;          // Used to pass arguments to the thread
    pthread_t thread;
    int pid;                  //Holds child process id when forking
    int error = 0;
    // Check arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
        exit(0);
    }
    signal(SIGPIPE, SIG_IGN);
    // Create a listening descriptor
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port);
    while (1) {
        error = 0;
        clientlen = sizeof(clientaddr);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        args = malloc(sizeof(arguments));
        args->connfd = connfd;
        args->clientaddr = clientaddr;
        args->playerID = threadCount;
        if (pthread_create(&thread, NULL, &handleConnection, args)) {
            perror("Thread creation failed!");
            continue;
        }

        if (threadCount == 1) {     // Once there are two clients,
            pid = fork();           // Create a new process to handle the next two
            if (pid != 0) {
                printf("In Parent process, created child: %d\n", pid);
                while(1){
                    sleep(1000000000000000000);  // Sleep while two child threads handle each connection
                }
            } else {
                threadCount = 0;                // Reset thread count in new process
                //continue while loop
            }
        }
    }
    // Control never reaches here
    exit(0);
}

void* handleConnection(void* argsVoid) {
    threadCount++;
    struct sockaddr_in clientaddr;  // Client address structure
    int connfd;                    // socket desciptor for talking with client
    int n;                         // General index and counting variables
    int playerID;                  // Player ID to identify whose turn is first
    char buf[256];
    int line;
    int selection;
    int error = 0;                 // Used to detect error in reading requests
    int retval=-1;
    arguments* args = (arguments*)argsVoid;  // Unpack all arguments passed into thread
    clientaddr = args->clientaddr;
    connfd = args->connfd;
    playerID = args->playerID;

    printf("Thread Number/Player ID: %d\n", threadCount);   // Confirm connection on server side
    while(threadCount != 2){        //Wait for next player to connect
        sleep(1);
    }
    if(playerID == 0){             // First player
        retval = write(args->connfd, "You are connected to a game! You are Player 1 and play X on the
board.", 256);    //Tell Client
        if(retval < 0){
            printf("Error writing!\n");
            pthread_exit(NULL);
        }
    }
    if(playerID == 1){             // Second Player
        retval = write(args->connfd, "You are connected to a game! You are Player 2 and play O on the
board. Please wait for Player 1 to make a move.", 256);   // Tell Client
        if(retval < 0){
            printf("Error writing!\n");
            pthread_exit(NULL);
        }
    }
    while (1) {
        if (turn > 8 && hodor == 0){
            retval = write(args->connfd, "Tie game!!!\n", strlen("Tie game!!!\n"));
            if (retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
            pthread_exit(NULL);
        }
        if (turn % 2 == 0 && playerID == 0){ // Player 1's turn
            if(hodor == 1){                  // Check if player has lost
                retval = write(args->connfd, "You lose!!!\n", strlen("You lose!!!\n"));
                if(retval < 0){
                    printf("Error writing!\n");
                    pthread_exit(NULL);
                }
                pthread_exit(NULL);
            }

            //Send the board to the player
            bzero(buf, 256);
            snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n",
board[0][0], board[0][1], board[0][2], board[1][0], board[1][1], board[1][2], board[2][0], board[2][1],
board[2][2]);
            retval = write(args->connfd, buf, 256);
            if(retval < 0){
                    printf("Error writing!\n");
                    pthread_exit(NULL);
            }
            if ((n = read(args->connfd, &selection, MAXLINE)) <= 0){
                error = 1;  //Used to fix a bug
                printf("process_request: client issued a bad request (1).\n");
                close(args->connfd);
                break;
            }
            // Parse the row and column of symbol from the user's selection
            int pos = selection;
            int row = --pos/3;
            int column = pos%3;
            if(board[row][column] == 'X' || board[row][column] == 'O'){     // Check if move is already in 2d
array
                retval = write(args->connfd, "Error, move already made!\n", strlen("Error, move already
made!\n"));
                if(retval < 0){
                    printf("Error writing!\n");
                    pthread_exit(NULL);
                }
                continue;
            }
            // Ensure pos is within range
            if (pos <0 || pos > 8){
                printf("\nPlease enter a proper value.\n");
                continue;
            }
            // Insert the symbol
        board[row][column] = 'X';

        if (error) {
           close(connfd);
           pthread_exit(NULL);
        }
        //check for winning line, diagonal
        if ((board[0][0] == board[1][1] && board[0][0] == board[2][2]) || (board[0][2] == board[1][1] &&
board[0][2] == board[2][0])){
              hodor = 1;      //Set winner signal value to 1
        }
        for (line = 0; line <= 2; line ++){
           if ((board[line][0] == board[line][1] && board[line][0] == board[line][2]) || (board[0][line] ==
board[1][line] && board[0][line] == board[2][line])){
              hodor = 1;      //Set winner signal value to 1
           }
        }
        //check for winning line, rows and columns

        if(hodor == 0){     // If there isn't a winner yet, print board
           bzero(buf,256);
           snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c |
%c\n\nWaiting for Player 2 to make a move\n", board[0][0], board[0][1], board[0][2], board[1][0],
board[1][1], board[1][2], board[2][0], board[2][1], board[2][2]);
           retval = write(args->connfd, buf, 256);
           if(retval < 0){
              printf("Error writing!\n");
              pthread_exit(NULL);
           }
        }else{           // If there is a winner, print final board state
           bzero(buf,256);
           snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n",
board[0][0], board[0][1], board[0][2], board[1][0], board[1][1], board[1][2], board[2][0], board[2][1],
board[2][2]);
           retval = write(args->connfd, buf, 256);
           if(retval < 0){
              printf("Error writing!\n");
              pthread_exit(NULL);
           }
        }

        if (error) {
           close(connfd);
           pthread_exit(NULL);
        }
        if (hodor==1){     // If you have won, send to client
           retval = write(args->connfd, "You win!!!\n", strlen("You win!!!\n"));
           if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
            turn++;
            pthread_exit(NULL);
        }
        turn++; //Increment turn
    }else if (turn % 2 == 1 && playerID == 1){      //Same code as above, for other player (swap X for O)
        if(hodor == 1){
            retval = write(args->connfd, "You lose!!!\n", strlen("You lose!!!\n"));
            if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
            pthread_exit(NULL);
        }
        bzero(buf, 256);
        snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n",
board[0][0], board[0][1], board[0][2], board[1][0], board[1][1], board[1][2], board[2][0], board[2][1],
board[2][2]);
        retval = write(args->connfd, buf, 256);
        if(retval < 0){
            printf("Error writing!\n");
            pthread_exit(NULL);
        }
        bzero(buf, 256);
        if ((n = read(args->connfd, &selection, MAXLINE)) <= 0){
            error = 1;  //Used to fix a bug
            printf("process_request: client issued a bad request (1).\n");
            close(args->connfd);
            break;
        }
        int pos = selection;
        int row = --pos/3;
        int column = pos%3;

        if(board[row][column] == 'X' || board[row][column] == 'O'){
            retval = write(args->connfd, "Error, move already made!\n", strlen("Error, move already
made!\n"));
            if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
            continue;
        }

        if (pos <0 || pos > 8){
            printf("\nPlease enter a proper value.\n");
            continue;
        }

        board[row][column] = 'O';

        if (error) {
            close(connfd);
            pthread_exit(NULL);
        }
        //check for winning line, diagonal
        if ((board[0][0] == board[1][1] && board[0][0] == board[2][2]) || (board[0][2] == board[1][1] &&
board[0][2] == board[2][0])){
                hodor = 1;
        }
        for (line = 0; line <= 2; line ++){
            if ((board[line][0] == board[line][1] && board[line][0] == board[line][2]) || (board[0][line] ==
board[1][line] && board[0][line] == board[2][line])){
                hodor = 1;
            }
        }
        //check for winning line, rows and columns
        if(hodor == 0){
            bzero(buf,256);
            snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c |
%c\n\nWaiting for Player 1 to make a move\n", board[0][0], board[0][1], board[0][2], board[1][0],
board[1][1], board[1][2], board[2][0], board[2][1], board[2][2]);
            retval = write(args->connfd, buf, 256);
            if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
        }else{
            bzero(buf,256);
            snprintf(buf, 256, "\n\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n ---+---+---\n %c | %c | %c\n",
board[0][0], board[0][1], board[0][2], board[1][0], board[1][1], board[1][2], board[2][0], board[2][1],
board[2][2]);
            retval = write(args->connfd, buf, 256);
            if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
        }

        if (error) {
            close(connfd);
            pthread_exit(NULL);
        }
        if (hodor==1){
            retval = write(args->connfd, "You win!!!\n", strlen("You win!!!\n"));
            if(retval < 0){
                printf("Error writing!\n");
                pthread_exit(NULL);
            }
            turn++;
            pthread_exit(NULL);
        }
        turn++; //Increment turn
    }else{
        sleep(1);
        continue;
    }
}
close(args->connfd);
return NULL;
}
```

T3Client.c

```
#include <stdio.h>
#include <stdlib.h>

#include <netdb.h>
#include <netinet/in.h>

#include <string.h>
#include "csapp.h"

int main(int argc, char *argv[]) {
   int sockfd, portno, n;
   struct sockaddr_in serv_addr;
   struct hostent *server;
   int selection;
   char buffer[256];

   if (argc < 3) {
      fprintf(stderr,"usage %s hostname port\n", argv[0]);
      exit(0);
   }

   portno = atoi(argv[2]);

   // Create a socket
   sockfd = Socket(AF_INET, SOCK_STREAM, 0);

   if (sockfd < 0) {
      perror("ERROR opening socket");
```

```
    exit(1);
  }

  server = Gethostbyname(argv[1]);

  if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
  }

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
  serv_addr.sin_port = htons(portno);

  // Now connect to the server
  if (connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    perror("ERROR connecting");
    exit(1);
  }

  bzero(buffer, 256);
  n = read(sockfd, buffer, 256);
  if(n < 0){
    printf("Error reading!\n");
  }
  printf("%s\n", buffer);

  // Now ask for a message from the user, this message will be read by server

        while(1){
    bzero(buffer, 256);
    n = read(sockfd, buffer, 256);
    if(n < 0){
      printf("Error reading!\n");
    }

    //Check if you have won, lost, or drawn

    printf("%s\n", buffer);
    if(strcmp(buffer, "You win!!!\n") == 0){
      exit(0);
    }else if(strcmp(buffer, "You lose!!!\n") == 0){
      exit(0);
    }else if(strcmp(buffer, "Tie game!!!\n") == 0){
      exit(0);
    }
    //Select the square you want to place your symbol in
    printf("\nPlease enter the number of the square:\n");
    scanf("%d", &selection);
    if (selection <1 || selection > 9){
      printf("\nPlease enter a proper value.\n");
      continue;
    }

    // Send message to the server
    n = write(sockfd, &selection, sizeof(selection));
    if (n < 0) {
      perror("ERROR writing to socket");
      exit(1);
    }

    // Now read server response
    bzero(buffer,256);
    n = read(sockfd, buffer, 256);
    if (n < 0) {
      perror("ERROR reading from socket");
      exit(1);
    }

    printf("%s\n",buffer);
  }
  return 0;
}
```