

ryan_morris_1550957

by Ryan Morris

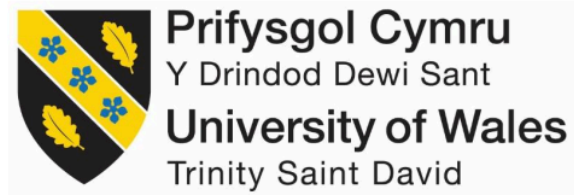
Submission date: 15-May-2025 02:34PM (UTC+0100)

Submission ID: 259387796

File name: 41185_Ryan_Morris_ryan_morris_1550957_664977_1658767003.docx (291.3K)

Word count: 1346

Character count: 7029



Socket Programming and Concurrency
ACCA6028

Course Work 2

Two Player Hangman Game Using TCP
Sockets in C++

Ryan Morris

1550957@student.uwtsd.ac.uk

15th May 2025

Table of Contents

INTRODUCTION.....	2
REQUIREMENTS AND DESIGN.....	2
SOCKET PROGRAMMING APPROACH	3
MESSAGE HANDLING AND PROTOCOL.....	4
UI, LIMITATIONS AND FUTURE IMPROVEMENTS	5
CONCLUSION	6

Introduction

For this assignment a networked two-player hangman game has been developed using C++ and TCP sockets. This report will document the design and implementation of the application. The game works by allowing the assigning the first connected client Player A, who can then set a word and a hint. The second client is assigned Player B is the guesser and will need to guess the word within a set number of guesses. The application was built in Visual Studio 2022 IDE as two separate projects, a client and a server, when the server executable is run, two clients need to be initialised to begin the game. Multiple clearly defined message classes have been implemented, which allow structured communication, good code readability and modularity.

Requirements and Design

The main functional requirement was to develop a two-player hangman game using the principles of socket programming in C++. Key functionality includes supporting player interaction over a network, with the server listening for connections and distinguishes the first client from the second client, the word setter and the guesser. Standard hangman rules were enforced, such as only guessing one letter per guess, not guessing the same letter multiple times, only being allowed a limited number of incorrect guesses and displaying the current game state after each guess. Both players should be able to see the current process of the game, with Client A being able to watch Client B's progress in real-time.

To achieve these functional requirements, a standard client-server architecture was implemented using TCP sockets. The server handles functions like game logic and synchronisation, assigning roles, validating guesses and providing game status updates. The client application then connects to the server and receives a role, first connection – Client A, second – Client B, and is prompted to send messages based on the role that they receive, the client application is also responsible for the game board.

To increase readability and maintainability, different functions such as networking, messaging and game flow are separated into separate classes, and time permitting will be separated further in VS filters (virtual folders). All communication between the server and clients is controlled by message classes, such as AssignRoleMessage, GuessLetterMessage, GuessResultMessage amongst others. These message classes work to serialise and unserialise data passed between endpoints. Making separate classes for separate functions created modularity, which helps to make extending the program or debugging it more efficient.

A simple terminal based UI was used to ensure a consistent user experience, the UI provides visual feedback through a simple ASCII based hangman gallows image and masked word updates. This helps keep users engaged and provides clarity as to what stage of the game they are at.

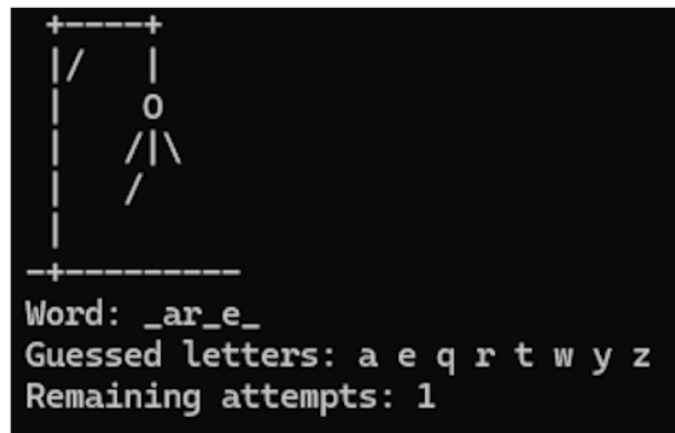
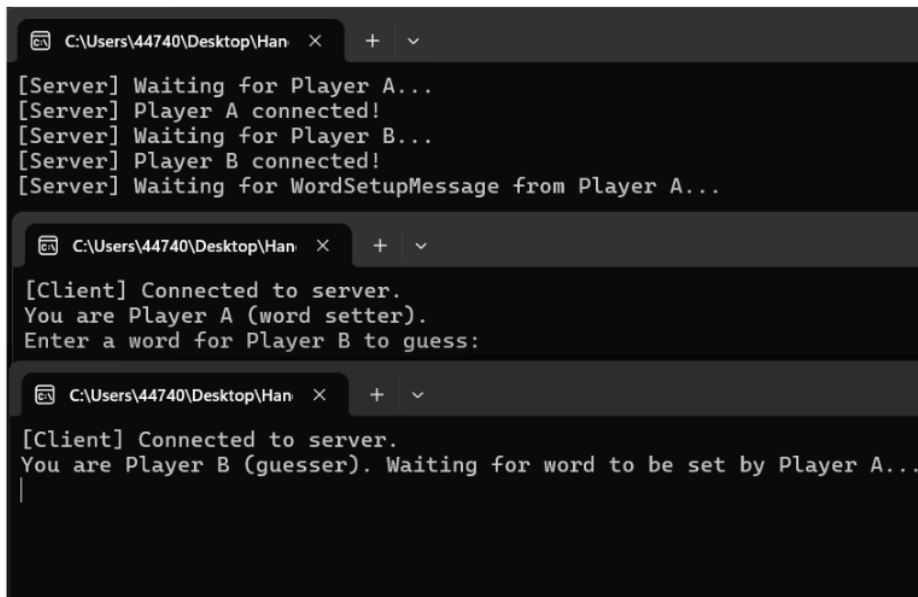


Figure 1 - Client A & B see this during guessing phase

Socket Programming Approach

This implementation of hangman utilises TCP sockets to let two players connect and play over a network, with one user setting the word to be guessed, and the other being the guesser. The server is to be run first and listens for two client connections, the first client to connect is assigned Player A, and the second Player B.



```
C:\Users\44740\Desktop\Han > [Server] Waiting for Player A...
C:\Users\44740\Desktop\Han > [Server] Player A connected!
C:\Users\44740\Desktop\Han > [Server] Waiting for Player B...
C:\Users\44740\Desktop\Han > [Server] Player B connected!
C:\Users\44740\Desktop\Han > [Server] Waiting for WordSetupMessage from Player A...

C:\Users\44740\Desktop\Han > [Client] Connected to server.
C:\Users\44740\Desktop\Han > You are Player A (word setter).
C:\Users\44740\Desktop\Han > Enter a word for Player B to guess:

C:\Users\44740\Desktop\Han > [Client] Connected to server.
C:\Users\44740\Desktop\Han > You are Player B (guesser). Waiting for word to be set by Player A...
C:\Users\44740\Desktop\Han > |
```

Figure 2 - Server assigning roles

The Windows Socket library, Winsock, was used to create a standard TCP socket. Once a connection is made, the client receives a message telling them their role, from then each client behaves according to that specific role, with Player A setting a word, and Player B put into a wait state until the word to guess is ready.

Using custom message classes has simplified handling communication. For example, when Player B guesses a letter, the guess is sent in a `GuessLetterMessage` to the server which checks the validity of the guess and responds using `GuessResultMessage`, this updates the masked word if necessary, as well as the number of guesses left. If this number is 0, it also informs if the player has lost or won. This approach helped keep communication between server and client clear and manageable.

Due to the two-player synchronous nature of the game, threading and concurrency weren't necessary. This contributed to the simplicity and kept the project easier to manage and test.

Message Handling and Protocol

As mentioned previously, communication between server and the two clients is handled with custom message classes. The base `Message` class is inherited by different message types, such as `AssignRoleMessage`, `GuessLetterMessage` and `GuessResultMessage`. The various messages are identified by an enum stored in the `Message` class. This method makes it easier to handle different aspects of the game, such as assigning roles or sending guesses.

```

MessageType type = static_cast<MessageType>(tType);

Message* m = nullptr;

switch (type)
{
case wordsetup:
    m = new WordSetupMessage();
    break;
case guessletter:
    m = new GuessLetterMessage();
    break;
case guessresult:
    m = new GuessResultMessage();
    break;
case assignrole:
    m = new AssignRoleMessage();
    break;
case readytoplay:
    m = new ReadyToPlayMessage();
    break;
default:
    std::cerr << "[Message] Unknown MessageType: " << tType << std::endl;
    return nullptr;
}

```

Figure 3 - Switch Statement for Message Types

To enable data to be sent across the network, each message is first converted to a string using a `Serialise()` function, the receiving end then uses `Unserialise()` to convert the string back into the correct message object. This process happens on both the server and the client side, when Player B guesses a letter for example, `GuessLetterMessage` is sent to the server. The server recognises the type by referring to enum value in the `Message` class using a switch statement and casts it to the relevant class to be processed. By doing this, long if else statements were avoided, and it also maintains modularity. Using this approach makes it easier to extend the games message types/functionality if required.

```

GuessLetterMessage guess(input[0]);
stringstream out;
guess.Serialise(out);
connection.SendString(out.str());

```

Figure 4 - Serialise data, snippet from `HangmanClient.cpp`

```

string raw = playerB.ReceiveString();
Message* guessBase = Message::Unserialise(raw);
if (!guessBase || guessBase->GetType() != guessletter)

```

Figure 5 - Unserialise data, snippet from `HangmanServer.cpp`

UI, Limitations and Future Improvements

The games interface was kept intentionally minimal, with just a text-based terminal used for both players. While simple, it provides users with clear prompts and game state updates, this includes ASCII hangman gallows visual, which shows how many

remaining guesses remain. A list of guessed letters and a masked word (e.g. "t_st" for "test" once 't' and 's' have been guessed), make the game easier to follow and more engaging. The game layout has been tested with various word lengths, guessing multiple letters at once, and was adjusted so both clients can view game progress in real-time.

Player B is initially entered into a waiting state, allowing Player A to set a word and hint, Player A has no input after the initial setting, but can watch Player B try to guess their word. Game logic to prevent guessing the same letter twice, or to enter more than one letter has been implemented to improve the robustness of the application. Some game behaviour needs to be addressed, such as checking to ensure Player A doesn't simply enter an empty string, time restraints mean this is needed in a future iteration. A small issue with hints means for the current version, they are not being displayed for the client, this too will need to be addressed.

A key limitation of the game is that there is only one round, and users can't select their own roles. A good improvement would be to allow rematches with alternating roles, coupled with the server tracking the scores to further improve engagement, perhaps with a best-of-three option. Another limitation is that there's no check on the word submitted by Player A, currently any string is accepted, a dictionary check could be implemented in the future to ensure it is a valid word.

Although concurrency and multithreading were not required for this simple version of the game, if a future version were to allow more players to play concurrent games, it would have to be considered.

Conclusion

This hangman project has successfully delivered a two-player game which can be played over a network using TCP sockets in C++. Although it's a simple version of the game, the core principles of client-server communication, message handling and game state sharing have been applied. The code is modular throughout, and due to the modularity, any future work could be implemented with relative ease.

FINAL GRADE

GENERAL COMMENTS

A good effort - a relatively simple application, but executed well, and a good deployment of principles.

70/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6