Multiclient 20 Questions

Caleb Loring, Ben Kollmar

Overview

Multiclient 20 Questions is a multi-user network application game. This game was chosen as it suited the project well and is a fun game that can be played with friends. In our version of 20 questions, there are 2 players: the player who chooses the object, the answerer, and the player who asks the questions, the questioner. To start the game, the answerer chooses an object and then describes its category: person, place, or thing. The questioner is then told the category of the object and starts asking questions in a yes/no format. After each question asked, the answerer responds to the question, with the questioner getting 20 questions to determine the object. If the questioner guesses the object at any time, they win the game. If all 20 questions are asked and the questioner has not guessed the object, the game ends. This game is played in the terminal by two players on the same network. Both players can connect to a locally hosted server and play with the same client.

System Architecture

This 20-question application is built on the TCP/IP protocol stack. It is composed of 3 components: the client, the main server, and the game logic. All three aspects communicate with each other to form the implementation of the game.

The client is the most basic of these components, stored in the `newClient` class. This client is a terminal-based application that allows players to connect to the server to play the game. To run the client, the port number the player wishes to connect on must be specified. The client connects to local servers by establishing a TCP socket connection, which supports a two-way communication channel. The clients remain active throughout the entire game to allow for instant communication between players.

The main server is used to initialize and set up game lobbies so that multiple games can run concurrently. Clients can connect to the main server on port 9999 using its TCP Socket. When a client connects to the main server, they are prompted to input a port number. Once the server verifies that the input port is valid for a new game lobby, it will create a new thread and launch a game lobby with a listener on the specified port. Then, players can run the client on the specified port to play against one another. This implementation ensures that each game is played on its own thread and port, such that no data handling errors occur. Game lobbies can only be created by the main server.

All the game logic is stored in the `gameLogic` class. This class essentially acts as a standalone server for each lobby, which entirely handles individual games. When the main server launches a new game lobby, this class creates a new TCP `ServerSocket`, which clients can connect to. Once two clients connect to the server on the specified port, input/output streams are established with both clients. Once these data streams are created, the game begins. Throughout the course of the game, data is sent back and forth between the server and each client. Data always flows through the server. Once either the questioner guesses the object or runs out of questions, both clients are notified that the game is over, and all sockets are closed. The port once again becomes available for lobbies to be created on.

The interactions of each of these components can be visualized as below. **Figure 1** demonstrates how lobby initialization occurs. The main server must be running for the game to be playable. First, a client connects to the main server by specifying port "9999" in the CLI. The client and server establish a connection through the TCP three-way handshake. Next, the server sets up a data input stream for the client and prompts the client to enter a valid port number (to act as the lobby code). If a valid port is read by the server, it creates a thread and launches a lobby which listens on the specified port.

Once the lobby is launched, clients can connect by running the client in the terminal with the given port number. **Figure 2** demonstrates how the actual game runs. The first client to connect is assigned as the answerer; as seen by the client which connects at t7, wherein the same TCP handshake occurs. At t8, the second client connects with the same handshake, being assigned the questioner role. Once both clients are connected, the `gameLogic` server creates data streams for both clients and prompts the answerer to enter their object. When the answerer does so, the message is sent back to the server, and the process is repeated with the category. Then, the category is sent to the questioner along with the prompt to ask a question. When the questioner enters their question, it is sent back to the server, then sent to the answerer. The answerer picks a response, which gets sent to the server and then to the client. This process is repeated until the game ends.

The figures demonstrate just one game server running. However, there can be many game servers running at once, each allowing 2 clients to connect at a time. The process for establishing a connection and starting the game remains the same in all cases. When the main server is shut down, all sockets from each game server close as well.

The 20 Questions game resides entirely in the application layer as shown in **Figure 3**. `Socket`, `PrintWriter`, and `BufferedReader` are all examples of application-level logic. At the application layer, messages are sent and received from the layers below, then processed. The game relies on TCP in the transport layer to create the required reliable communication channel. The transport layer provides logical communication between client and server, containing both a sender and a receiver. The sender separates messages into segments, while the receiver reassembles segments and passes the data up. The network layer sender receives the segments from above and encapsulates them as datagrams. The IP header contains info that the routers use to determine how to get the datagram to the receiving host. The link and physical layers actually deliver the message data from host to receiver. These 5 layers work together to ensure that messages can reliably be sent from client to server, and server to client.

Figure 1

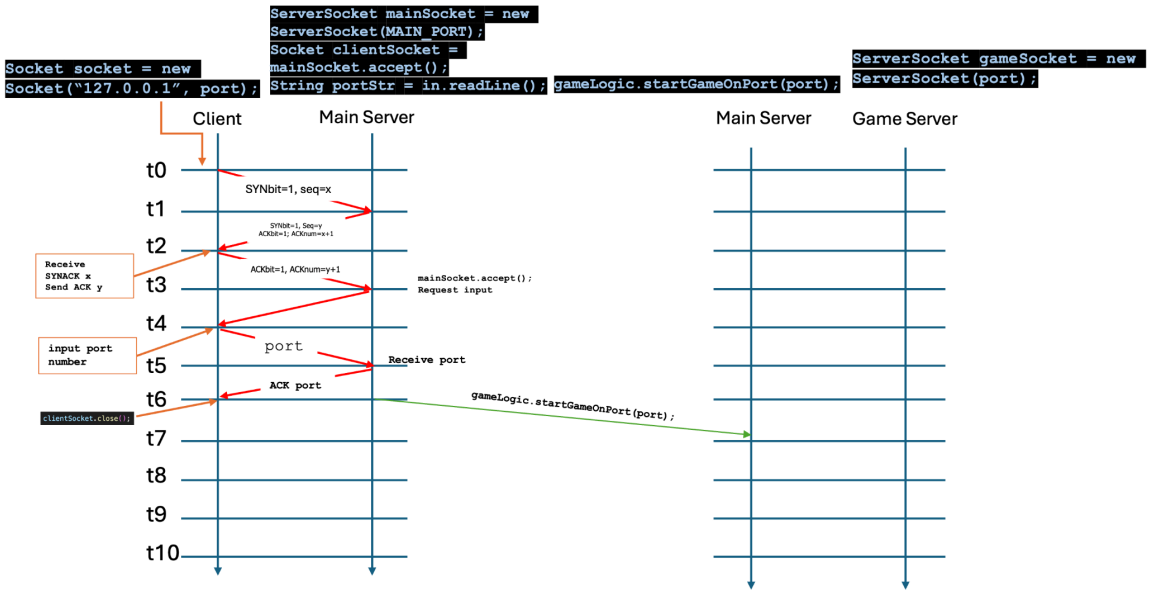


Figure 2

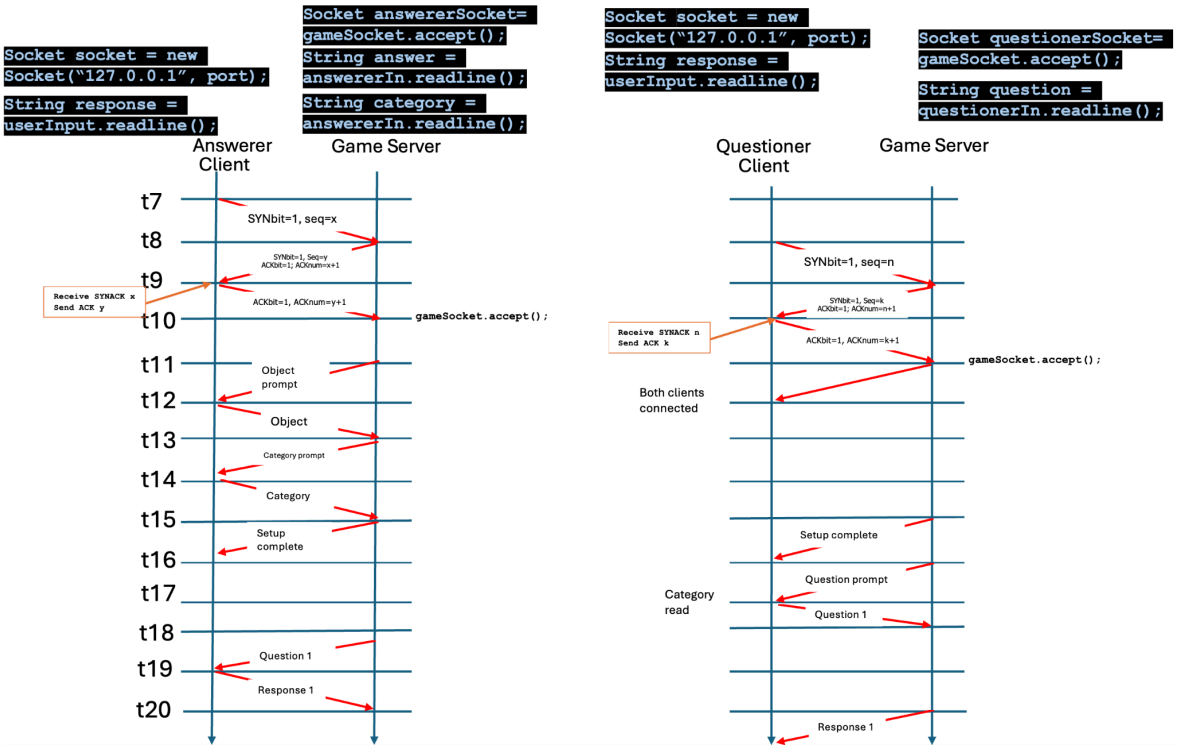
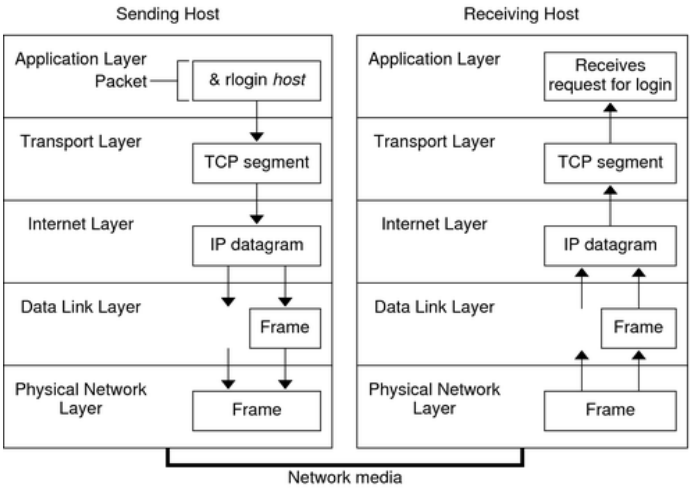


Figure 3



System Implementation

The multiplayer 20-Questions game application is developed entirely in the Java programming language. The source code is organized into three components: a lobby server `mainServer`, the core game logic `gameLogic`, and the client interface `newClient`. The implementation exclusively uses standard Java

libraries, avoiding the need for any third-party dependencies. Specifically, the application relies on the java.io package for input/output operations, the java.net package for managing TCP socket connections between the server and clients, and the java.util.concurrent package for managing concurrency on the server side through thread pools. In terms of platform support, the application is platform-independent and can run on any system that supports Java, including Windows, macOS, and Linux. Since the system is entirely console-based and does not rely on platform-specific APIs or graphical interfaces, no additional configurations are required for cross-platform compatibility.

To execute the application, users must first compile the Java source files using the Java compiler: "`javac mainServer.java newClient.java gameLogic.java`". Once compiled, the server is started by running the `mainServer` class: "`java mainServer`". This initializes a listening service on port 9999 to handle lobby creation requests. Clients can then be launched from separate terminal sessions using: "`java newClient 9999`". This connects the client to the main lobby server, where they can create a game session. Upon specifying an available port, the server will launch a new game instance on that port. Players then connect to the newly designated port using the same `newClient` application to begin interaction.

Conclusion

This project allowed us to experience the real process of creating a network application. It has enriched our understanding of how the network layer functions. This application is not just a simple single-server game that 2 players can play on, the multi-server implementation takes this project beyond the basics. The application is designed to simplify the experience for the clients, allowing as many people to play as desired. Some possible enhancements include creating a play again option, or the opportunity to swap roles before the game setup occurs. Overall, this project has been a great learning experience that has allowed us to create a multi-user network application.

References

How the TCP/IP Protocols Handle Data Communications. (2011, August 1). https://docs.oracle.com/cd/E23823_01/html/816-4554/ipov-29.html

Siddiqui, Farhan. *Networks 3-1*. 2025, Personal class material.

Siddiqui, Farhan. *Networks 3-6*. 2025, Personal class material.

Siddiqui, Farhan. *Networks 4-1*. 2025, Personal class material.