Net Centric Lab 4

Nguyen Manh Viet Khoi
ITCSIU21081

Server.go

```go
package main

import (
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "fmt"
    "log"
    "math/rand"
    "net"
    "os"
    "strings"
    "time"
)

const (
    HOST               = "localhost"
    PORT               = "8080"
    TYPE               = "tcp"
    TIMEOUT            = 30 * time.Second
    POINTS_PER_LETTER  = 10
    MIN_PLAYERS        = 2
    MAX_ATTEMPTS       = 6
    TURN_TIMEOUT       = 30 * time.Second
    WORDS_FILE         = "words.json"
)

type User struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

type Game struct {
    Word        string
    Revealed    []bool
    Players     []*Player
    Attempts    int
    MaxAttempts int
```

```
    Completion  int
}

type Player struct {
    Conn   net.Conn
    Name   string
    Score  int
    IsTurn bool
}

type WordsData struct {
    Words []string `json:"words"`
}

func main() {
    rand.Seed(time.Now().UnixNano())
    words, err := loadWordsFromFile(WORDS_FILE)
    if err != nil {
        log.Fatal("Error loading words:", err)
    }
    word := words[rand.Intn(len(words))]

    listen, err := net.Listen(TYPE, HOST+":"+PORT)
    if err != nil {
        log.Fatal(err)
    }
    defer listen.Close()

    fmt.Println("Guessing game server started. Hidden word:", word)

    var players []*Player

    for {
        conn, err := listen.Accept()
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("Client connected from", conn.RemoteAddr())

        // Authentication
        if !authenticate(conn) {
            fmt.Println("Authentication failed. Closing connection.")
            conn.Write([]byte("authentication_failed\n"))
            conn.Close()
            continue
        }

        conn.Write([]byte("authenticated\n"))
```

```
        fmt.Println("Authentication successful.")

        // Add player to the list
        player := &Player{
            Conn: conn,
            Name: fmt.Sprintf("Player_%s", conn.RemoteAddr().String()),
        }
        players = append(players, player)

        // Notify players to wait if not enough players
        if len(players) < MIN_PLAYERS {
            player.Conn.Write([]byte("Waiting for other players to join...\n"))
            continue
        }

        // Start the game
        game := &Game{
            Word:        word,
            Revealed:    make([]bool, len(word)),
            Players:     players,
            MaxAttempts: MAX_ATTEMPTS,
        }
        runGame(game)
        break // Exit loop after starting the game
    }
}

func authenticate(conn net.Conn) bool {
    authData := readFromConn(conn)
    parts := strings.Split(authData, "_")
    if len(parts) != 2 {
        return false
    }

    username := parts[0]
    receivedPassword := parts[1]

    users, err := loadUsersFromFile("users.json")
    if err != nil {
        log.Println("Error loading users:", err)
        return false
    }
    receivedHashedPassword := hashPassword(receivedPassword)
    for _, user := range users {
        if user.Username == username && hashPassword(user.Password) ==
receivedHashedPassword {
            return true
        }
```

```
    }

    return false
}

func loadWordsFromFile(filename string) ([]string, error) {
    file, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    var wordsData WordsData
    err = json.NewDecoder(file).Decode(&wordsData)
    if err != nil {
        return nil, err
    }

    return wordsData.Words, nil
}

func readFromConn(conn net.Conn) string {
    buffer := make([]byte, 1024)
    n, err := conn.Read(buffer)
    if err != nil {
        log.Println("Error reading:", err)
        return ""
    }
    return strings.TrimSpace(string(buffer[:n]))
}

func loadUsersFromFile(filename string) ([]User, error) {
    file, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    var users struct {
        Users []User `json:"users"`
    }
    err = json.NewDecoder(file).Decode(&users)
    if err != nil {
        return nil, err
    }

    return users.Users, nil
}
```

```
func hashPassword(password string) string {
    hasher := sha256.New()
    hasher.Write([]byte(password))
    return hex.EncodeToString(hasher.Sum(nil))
}

func runGame(game *Game) {
    currentPlayerIndex := 0

    for game.Completion < len(game.Word) && game.Attempts < game.MaxAttempts {
        currentPlayer := game.Players[currentPlayerIndex]
        currentPlayer.IsTurn = true

        currentPlayer.Conn.Write([]byte(fmt.Sprintf("Your turn! Hidden word: %s\n",
revealWord(game))))

        // Set timeout for turn
        timer := time.NewTimer(TURN_TIMEOUT)

        select {
        case <-timer.C:
            log.Printf("%s's turn timed out.\n", currentPlayer.Name)
            currentPlayer.Conn.Write([]byte("Timeout! You lost your turn.\n"))
            currentPlayer.IsTurn = false
            game.Attempts++
            currentPlayerIndex = (currentPlayerIndex + 1) % len(game.Players)
            continue
        case <-time.After(10 * time.Millisecond):
            // Check if client sent any input
            guess := readFromConn(currentPlayer.Conn)
            if len(guess) != 1 {
                currentPlayer.Conn.Write([]byte("Invalid input. Guess one letter at a
time.\n"))
                continue
            }

            correct := processGuess(guess[0], game)
            if correct {
                currentPlayer.Score += POINTS_PER_LETTER
                currentPlayer.Conn.Write([]byte(fmt.Sprintf("Correct guess! Score: %d\
n", currentPlayer.Score)))
            } else {
                currentPlayer.Conn.Write([]byte("Incorrect guess.\n"))
            }

            if game.Completion == len(game.Word) {
                endGame(game, currentPlayer)
```

```
                return
            }
        }
        currentPlayer.IsTurn = false
        currentPlayerIndex = (currentPlayerIndex + 1) % len(game.Players)
    }
    endGame(game, nil) // Game over due to max attempts
}

func processGuess(letter byte, game *Game) bool {
    correct := false
    for i, char := range game.Word {
        if byte(char) == letter && !game.Revealed[i] {
            game.Revealed[i] = true
            game.Completion++
            correct = true
        }
    }
    return correct
}

func revealWord(game *Game) string {
    revealedWord := ""
    for i, char := range game.Word {
        if game.Revealed[i] {
            revealedWord += string(char)
        } else {
            revealedWord += "_"
        }
    }
    return revealedWord
}

func endGame(game *Game, winner *Player) {
    for _, player := range game.Players {
        if winner != nil && player == winner {
            player.Conn.Write([]byte(fmt.Sprintf("Congratulations! You guessed the
word: %s\n", game.Word)))
        } else {
            player.Conn.Write([]byte(fmt.Sprintf("Game over! Hidden word was: %s\n",
game.Word)))
        }
        player.Conn.Close()
    }
}
```

Client.go

```go
package main

import (
    "bufio"
    "fmt"
    "log"
    "net"
    "os"
    "strings"
)

const (
    HOST = "localhost"
    PORT = "8080"
)

func main() {
    conn, err := net.Dial("tcp", HOST+":"+PORT)
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    reader := bufio.NewReader(conn)
    fmt.Println("Connected to Hangman Game Server")

    // Authentication
    username := prompt("Enter username: ")
    password := prompt("Enter password: ")

    authData := fmt.Sprintf("%s_%s\n", username, password)
    conn.Write([]byte(authData))

    response, err := reader.ReadString('\n')
    if err != nil {
        log.Fatal(err)
    }
    if strings.TrimSpace(response) != "authenticated" {
        fmt.Println("Authentication failed.")
        return
    }
    fmt.Println("Authentication successful. Starting game.")

    // Game loop
    for {
        message, err := reader.ReadString('\n')
```

```
        if err != nil {
            log.Fatal(err)
        }
        fmt.Print(message)

        if strings.Contains(message, "Your turn") {
            guess := prompt("Enter your guess (one letter): ")
            conn.Write([]byte(guess + "\n"))
        }

        if strings.Contains(message, "Game over") {
            break
        }
    }
}

func prompt(promptMsg string) string {
    fmt.Print(promptMsg)
    reader := bufio.NewReader(os.Stdin)
    input, err := reader.ReadString('\n')
    if err != nil {
        log.Fatal(err)
    }
    return strings.TrimSpace(input)
}
```

Users.json

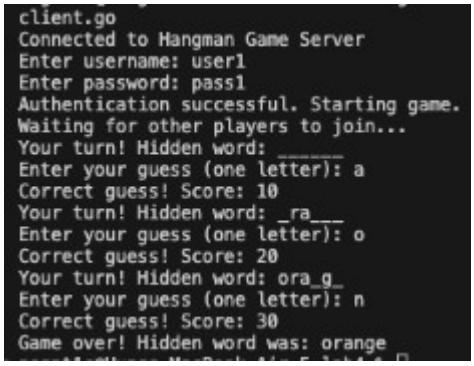
```
{
  "users": [
    {
      "username": "user1",
      "password": "pass1"
    },
    {
      "username": "user2",
      "password": "pass2"
    }
  ]
}
```

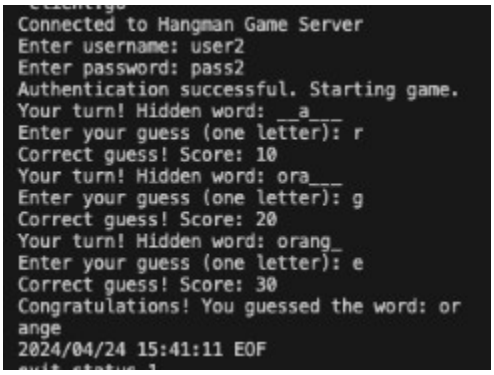
Words.json

```
{
    "words": [
```

```
        "apple",
        "banana",
        "cherry",
        "orange"
    ]
}
```

Output:





Here how it works:

Game Setup

1. **Word Selection**: The server randomly selects a word from a predefined list of words (stored in **words.json**).
2. **Player Connection**: Players connect to the server using a TCP socket and authenticate themselves by sending their username and password in the format "username_password".
3. **Game Initialization**: Once the required number of players (specified in **MIN_PLAYERS**) have connected and authenticated, the game starts.

Gameplay

1. **Hidden Word**: The server hides the selected word and represents it with underscores (_) for unrevealed letters.

2. **Turn-Based Gameplay**: Players take turns guessing letters to reveal the hidden word.
3. **Guess Mechanism**:
    - Players send single-letter guesses to the server.
    - The server checks the guess against the hidden word and reveals any matching letters.
    - Correct guesses increase the player's score based on the number of occurrences of the guessed letter in the word.
4. **Timeout Handling**:
    - Each player has a time limit (specified in **TURN_TIMEOUT**) for their turn.
    - If a player fails to guess within the time limit, they lose their turn, and the next player plays.
5. **Game End**:
    - The game continues until a player correctly guesses the entire word or until the maximum attempts (specified in **MAX_ATTEMPTS**) are reached.
    - If a player guesses the entire word correctly, they win the game.
    - If the maximum attempts are reached without guessing the word, the game ends, and the players can start a new game.

Customization Options

- **Word List**: You can modify the **words.json** file to include different words for the game.
- **Game Parameters**: Constants like **POINTS_PER_LETTER**, **MAX_ATTEMPTS**, and **TURN_TIMEOUT** can be adjusted to change game behavior and difficulty.