Network Protocol Documentation

This document provides a comprehensive description of the client-server communication protocol implemented in this project. It details the JSON message structure, command types, authentication flow, error handling, and all possible communication scenarios.

Table of Contents

General Protocol Information

- **Transport Protocol**: TCP
- **Default Port**: 1337
- **Default Hostname**: localhost
- **Character Encoding**: UTF-8
- **Message Delimiter**: Newline character (`\n`)
- **Maximum Message Size**: 4096 bytes
- **Data Format**: JSON

Running the Application

Server

To run the server:

```
./ex1_server.py users_file [port] [--verbose]
```

- `users_file`: Path to file containing username/password pairs
- `port`: (Optional) Port number to listen on (default: 1337)
- `--verbose`: (Optional) Enable verbose logging

Client

To run the client:

```
./ex1_client.py [hostname [port]] [--verbose]
```

- `hostname`: (Optional) Server hostname (default: localhost)
- `port`: (Optional) Port number to connect to (default: 1337)
- `--verbose`: (Optional) Enable verbose logging

- Note: You cannot provide a port without also providing a hostname

Message Format

All messages exchanged between client and server follow this general structure:

```
{
  "type": "command_type",
  "param1": "value1",
  "param2": "value2",
  ...
}
```

Each message must: 1. Be valid JSON 2. Include a "type" field 3. End with a newline character (`\n`) 4. Contain all required parameters for the specific command type

Authentication Flow

The server implements a strict two-step authentication process:

1. **Username Submission**:
   - Client must first submit a username
   - No other commands are accepted before username submission
   - Attempting any other command results in immediate disconnection
2. **Password Submission**:
   - After submitting a valid username, client must submit password
   - No other commands are accepted at this stage
   - Attempting any other command results in immediate disconnection
3. **Authentication States**:
   - `0`: Not authenticated (initial state)
   - `1`: Username submitted, waiting for password
   - `2`: Fully authenticated, can access all commands

Command Types

Authentication Commands

1. `login_username`

- **Purpose**: Submit username for authentication
- **Direction**: Client → Server
- **Required When**: First message after connection
- **Format**:

  ```
  {
    "type": "login_username",
  ```

```
  "username": "username_value"
}
```

- **Possible Responses**:
  - `continue`: Username accepted, proceed to password
  - `login_failure`: Username not found

2. `login_password`

- **Purpose**: Submit password for authentication
- **Direction**: Client → Server
- **Required When**: After username accepted
- **Format**:

```
{
  "type": "login_password",
  "password": "password_value"
}
```

- **Possible Responses**:
  - `login_success`: Authentication successful
  - `login_failure`: Incorrect password

Functional Commands (Post-Authentication)

3. `lcm`

- **Purpose**: Calculate Least Common Multiple of two integers
- **Direction**: Client → Server
- **Format**:

```
{
  "type": "lcm",
  "x": integer_value,
  "y": integer_value
}
```

- **Possible Responses**:
  - `lcm_result`: Contains computed LCM
  - `error`: Invalid parameters

4. `parentheses`

- **Purpose**: Check if a string of parentheses is balanced
- **Direction**: Client → Server

- **Format**:

  ```
  {
    "type": "parentheses",
    "string": "parentheses_string"
  }
  ```

- **Note**: The string should only contain `(` and `)` characters
- **Possible Responses**:
  - `parentheses_result`: Result of the balance check (true/false)
  - `error`: Invalid parameters or string contains invalid characters

5. `caesar`

- **Purpose**: Apply Caesar cipher to a text string
- **Direction**: Client → Server
- **Format**:

  ```
  {
    "type": "caesar",
    "text": "text_to_encrypt",
    "shift": integer_value
  }
  ```

- **Note**: The text may only contain alphabetic characters and spaces
- **Possible Responses**:
  - `caesar_result`: Contains ciphertext
  - `error`: Invalid parameters or text contains invalid characters

Response Types

Server → Client Responses

1. `greeting`

- **Purpose**: Initial welcome message
- **Sent When**: Client first connects
- **Format**:

  ```
  {
    "type": "greeting",
    "message": "Welcome! Please log in."
  }
  ```

2. `continue`
    - **Purpose**: Indicate username was accepted
    - **Sent When**: Valid username submitted
    - **Format**:

    ```
    {
      "type": "continue",
      "message": ""
    }
    ```

3. `login_success`
    - **Purpose**: Indicate successful authentication
    - **Sent When**: Correct password submitted
    - **Format**:

    ```
    {
      "type": "login_success",
      "message": "Hi, {username}, good to see you."
    }
    ```

4. `login_failure`
    - **Purpose**: Indicate authentication failure
    - **Sent When**: Invalid username or password
    - **Format**:

    ```
    {
      "type": "login_failure",
      "message": "Failed to login."
    }
    ```

5. `error`
    - **Purpose**: Report errors in command processing
    - **Sent When**: Command has invalid format or parameters
    - **Format**:

    ```
    {
      "type": "error",
      "message": "Error description"
    }
    ```

6. `lcm_result`

- **Purpose**: Return LCM calculation result
- **Sent When**: Valid LCM calculation request processed
- **Format**:

  ```
  {
    "type": "lcm_result",
    "result": integer_value
  }
  ```

7. `parentheses_result`

- **Purpose**: Return parentheses balance check result
- **Sent When**: Valid parentheses check request processed
- **Format**:

  ```
  {
    "type": "parentheses_result",
    "result": boolean_value
  }
  ```

8. `caesar_result`

- **Purpose**: Return Caesar cipher result
- **Sent When**: Valid Caesar cipher request processed
- **Format**:

  ```
  {
    "type": "caesar_result",
    "result": "encrypted_text"
  }
  ```

Error Handling

The protocol implements several error handling mechanisms:

1. JSON Validation Errors

- Invalid JSON results in error response
- Format:

  ```
  {
    "type": "error",
    "message": "Invalid JSON format."
  }
  ```

2. Authentication Errors

- If a client attempts any command before successful authentication, the server immediately disconnects the client
- For invalid username/password, server responds with `login_failure`

3. Command Parameter Errors

- Missing or invalid parameters result in error response
- Format:

  ```
  {
    "type": "error",
    "message": "Error description"
  }
  ```

4. Function-Specific Errors

- Each function has specific validation requirements:
  - `lcm`: Parameters must be valid integers
  - `parentheses`: String must only contain parentheses characters
  - `caesar`: Text must only contain alphabetic characters and spaces

Connection and Disconnection

Connection Establishment

1. Client initiates TCP connection to server
2. Server accepts connection and sends greeting
3. Client must begin authentication process

Automatic Disconnection

The server will automatically disconnect a client in the following situations: 1. Client sends any command other than `login_username` when in authentication state 0 2. Client sends any command other than `login_password` when in authentication state 1 3. Socket error or exception occurs

Client Disconnect

The client can disconnect at any time by closing the socket connection.

Complete Communication Flows

Successful Authentication Flow

```
Client connects to server
Server → Client: {"type": "greeting", "message": "Welcome! Please log in."} + \n
```

```
Client → Server: {"type": "login_username", "username": "valid_user"} + \n
Server → Client: {"type": "continue", "message": ""} + \n
Client → Server: {"type": "login_password", "password": "correct_password"} + \n
Server → Client: {"type": "login_success", "message": "Hi, valid_user, good to see you."} +
```

Failed Authentication Flow (Invalid Username)

```
Client connects to server
Server → Client: {"type": "greeting", "message": "Welcome! Please log in."} + \n
Client → Server: {"type": "login_username", "username": "invalid_user"} + \n
Server → Client: {"type": "login_failure", "message": "Failed to login."} + \n
```

Failed Authentication Flow (Invalid Password)

```
Client connects to server
Server → Client: {"type": "greeting", "message": "Welcome! Please log in."} + \n
Client → Server: {"type": "login_username", "username": "valid_user"} + \n
Server → Client: {"type": "continue", "message": ""} + \n
Client → Server: {"type": "login_password", "password": "wrong_password"} + \n
Server → Client: {"type": "login_failure", "message": "Failed to login."} + \n
```

Authentication Protocol Violation (Disconnect)

```
Client connects to server
Server → Client: {"type": "greeting", "message": "Welcome! Please log in."} + \n
Client → Server: {"type": "lcm", "x": 4, "y": 6} + \n
Server disconnects client
```

Successful LCM Calculation

```
(After successful authentication)
Client → Server: {"type": "lcm", "x": 4, "y": 6} + \n
Server → Client: {"type": "lcm_result", "result": 12} + \n
```

Successful Parentheses Check

```
(After successful authentication)
Client → Server: {"type": "parentheses", "string": "((()))"} + \n
Server → Client: {"type": "parentheses_result", "result": true} + \n
```

Successful Caesar Cipher

```
(After successful authentication)
Client → Server: {"type": "caesar", "text": "hello", "shift": 3} + \n
Server → Client: {"type": "caesar_result", "result": "khoor"} + \n
```

Invalid Command Format

```
(After successful authentication)
Client → Server: {"type": "lcm", "x": "not_a_number", "y": 6} + \n
Server → Client: {"type": "error", "message": "Invalid parameters for LCM."} + \n
```

Invalid Parentheses String

```
(After successful authentication)
Client → Server: {"type": "parentheses", "string": "((a))"} + \n
Server → Client: {"type": "error", "message": "String contains invalid characters."} + \n
```

Invalid Caesar Text

```
(After successful authentication)
Client → Server: {"type": "caesar", "text": "hello123", "shift": 3} + \n
Server → Client: {"type": "error", "message": "error: invalid input"} + \n
```

Implementation Notes

Message Processing Logic

1. **Server Message Processing**:
   - Messages are buffered until a complete newline-terminated message is received
   - Each complete message is processed individually
   - Response is immediately sent back to client
2. **Client Message Processing**:
   - Similar to server, client buffers incoming data until a complete message is received
   - Client interprets server responses and prompts for appropriate user input
   - User commands are validated before sending to server

Buffering and Message Boundaries

- Both client and server maintain separate read and send buffers
- Messages are processed only when complete (terminated with newline)
- Incomplete messages are kept in buffer until more data arrives
- Maximum message size is enforced (4096 bytes)

Client Input Handling

The client accepts these command formats from user input: - `User: username` - For username submission - `Password: password` - For password submission - `lcm: x y` - For LCM calculation - `parentheses: string` - For parentheses balance check - `caesar: text shift` - For Caesar cipher encryption - `quit` - To exit the application

Security Considerations

1. **Authentication**:
    - Two-step authentication process
    - We could not implement proper secure authentication since Username and Password were required to be passed seperatly…
    - Immediate disconnection for protocol violations
    - Clear separation of authentication states
2. **Input Validation**:
    - All inputs are validated before processing
    - Specific validation for each command type
    - Rejection of malformed or invalid inputs
3. **Error Handling**:
    - Clear error messages
    - No leakage of internal implementation details
    - Graceful handling of unexpected inputs