# Report on Implementation of Version 1

## 1 Overview

This report covers the implementation details of a client-server system designed to handle C++ source code submission, compilation, execution, and validation.

## 2 Server Code : server.cpp

### 2.1 File:

#### 2.1.1 Key Components

**Constants and Includes**

- **Constants**: Defines buffer size (`BUFFER_SIZE`) and response types (`PASS`, `COMPILER_ERROR`, `RUNTIME_ERROR`, `OUTPUT_ERROR`).
- **Includes**: Necessary libraries for socket communication and file operations.

**Debug Function**

- **Purpose**: To compile and execute the received C++ source code, and compare the output with the expected output.
- **Steps**:
  1. **Receive Source Code**: The server receives the source code from the client and writes it to a file named `server_test.cpp`.
  2. **Compile Source Code**: Compiles the source code using `g++`. If compilation fails, it logs the errors to `compile_error.txt` and sends a `"COMPILER ERROR"` response.
  3. **Execute Compiled Code**: Executes the compiled program and captures the output and errors in `output.txt`. If a runtime error occurs, it reads the error from `output.txt` and sends a `"RUNTIME ERROR"` response.
  4. **Compare Output**: Compares the output with the expected output using the `diff` command and stores the result in `diff_output.txt`. If the output does not match, it sends an `"OUTPUT ERROR"` response.

5. **Send Result**: If all checks pass, it sends a `"PASS"` response to the client.

**Main Function**

- **Purpose**: Initializes the server, listens for incoming client connections, and processes each request using the `debug` function.
- **Steps**:
  1. **Socket Creation**: Creates a socket and binds it to the specified port.
  2. **Listening and Accepting Connections**: Listens for incoming connections and accepts them.
  3. **Process Requests**: For each client connection, processes the received source code by calling the `debug` function.
  4. **Send Response**: Sends the result of the code evaluation back to the client.

# 3 Client Code : client.cpp

### 3.0.1 Key Components

**Constants and Includes**

- **Constants**: Defines buffer size (`BUFFER_SIZE`).
- **Includes**: Necessary libraries for socket communication and file operations.

**Main Function**

- **Purpose**: To send a C++ source code file to the server and receive the result.
- **Steps**:
  1. **Parse Server Address**: Extracts the server's IP address and port from the command-line arguments.
  2. **Socket Creation and Connection**: Creates a socket and connects to the server.
  3. **Send Source Code**: Reads the source code file and sends its contents to the server.
  4. **Signal End of File**: Shuts down the write side of the socket to indicate the end of the file.
  5. **Receive and Display Result**: Receives the server's response and prints it.

# Report for Version 2

## 1 Overview

Version 2 introduces several enhancements over Version 1, including improved performance metrics and multi-threaded server handling. This version includes a client that benchmarks the server's response time and throughput while the server processes incoming requests using multiple threads.

## 2 Implementation Details

### 2.1 Client

- **Functionality:** The client sends a C source code file to the server multiple times based on user input, measures the response time for each request, and calculates throughput metrics.
- **Parameters:**
  - `serverIP:port` - The server's IP address and port number.
  - `sourceCodeFileTobeGraded` - The file containing the C source code to be sent.
  - `numberofiteration` - Number of times the client will send the code to the server.
  - `sleeptime` - The time (in seconds) the client will wait between iterations.
  - `timeout` - The time (in seconds) the client will wait for a response before timing out.
- **Performance Metrics:** The client calculates and displays the total response time, average response time, and throughput (successful responses per second).

### 2.2 Server

- **Functionality:** The server receives C source code files, compiles them, executes the compiled code, and compares the output with expected results. It then sends the result back to the client.

- **Multi-threading:** The server uses threads to handle multiple client requests concurrently. Each thread processes a single client's request, allowing for simultaneous handling of multiple submissions.
- **Error Handling:** The server provides detailed feedback for compilation errors, runtime errors, and output mismatches, enhancing the debugging process for clients.

# 3 Results

## 3.1 Client Performance Metrics

- **Successful Responses:** The number of successful responses received from the server.
- **Total Response Time:** The sum of all response times across iterations.
- **Average Response Time:** The average time taken for a response, calculated in microseconds.
- **Throughput:** The rate of successful responses per second.

## 3.2 Server Performance

- The server successfully handles multiple client requests concurrently.
- It efficiently compiles and executes code, providing accurate results and feedback based on the execution and output comparison.

# 4 Summary

Version 2 demonstrates significant improvements in performance measurement and server handling over Version 1. The client now includes detailed timing metrics and throughput calculations, while the server can handle concurrent requests through multi-threading, resulting in improved efficiency and scalability.

# Report for Version 3

## 1 Overview

Version 3 of the client-server application includes improvements in handling client requests and server responses with more efficient thread management and enhanced performance metrics. The client sends a file to the server and receives the result, while the server processes requests using a thread pool.

## 2 Client

### 2.1 Functionality

The client connects to the server, sends a file, and receives feedback on the file's compilation and execution status.

### 2.2 Features

- **Timeout Handling**: Uses `alarm()` and `signal()` to handle timeouts and ensure responsiveness.
- **Performance Metrics**: Measures response time and throughput to assess server performance.
- **Usage**: `./client localhost <server_port> c_file numberofiteration sleeptime timeout`

### 2.3 Code Highlights

- Sets up the socket and handles connections.
- Measures response times and calculates average response time and throughput.
- Handles different response scenarios (`CERR`, `RERR`, `OERR`, `PASS`).

## 3 Server

### 3.1 Functionality

The server receives files from clients, processes them (compiles and executes), and sends back results.

### 3.2 Features

- **Thread Pool**: Utilizes a pool of threads to handle multiple client requests concurrently.
- **Queue Management**: Manages incoming requests using a queue with synchronization via mutexes and condition variables.
- **Usage**: `./server <port> <thread_pool_size>`

### 3.3 Code Highlights

- Handles client connections and file processing in separate threads.
- Executes commands to compile and run received code files, handling errors and sending appropriate messages back to the client.
- Maintains a queue to manage incoming requests and ensures efficient processing with multiple threads.

# 4 Performance Metrics

## 4.1 Client-Side

- **Successful Responses**: Total number of successful interactions with the server.
- **Response Time**: Time taken for server responses, averaged over multiple iterations.
- **Throughput**: Calculated as the number of successful responses divided by the total time.

## 4.2 Server-Side

- **Concurrency**: Improved handling of concurrent requests through a thread pool.
- **Queue Management**: Efficient request handling with queue-based request management.

# 5 Conclusion

Version 3 enhances the client-server interaction by incorporating performance metrics and efficient request handling. The client measures server response times and throughput, while the server leverages threading and queuing mechanisms to handle multiple requests effectively. These improvements aim to provide a more robust and scalable solution for file processing and server-client communication.