Load generating Client (in a loop)

```cpp
#include <iostream>
#include <fstream>
#include <cstring>
#include <chrono>
#include <pthread.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>

using namespace std;

const int BUFFER_SIZE = 1024;
const int MAX_FILE_SIZE_BYTES = 4;
const int MAX_TRIES = 5;

struct sockaddr_in serv_addr;
string file_path;

int send_file(int sockfd, string file_path)
{
    char buffer[BUFFER_SIZE];
    bzero(buffer, BUFFER_SIZE);
    FILE *file = fopen(file_path.c_str(), "rb");
    if (!file)
    {
        perror("Error opening file");
        return -1;
    }

    fseek(file, 0L, SEEK_END);
    int file_size = ftell(file);
    fseek(file, 0L, SEEK_SET);
    char file_size_bytes[MAX_FILE_SIZE_BYTES];
    memcpy(file_size_bytes, &file_size, sizeof(file_size));
    if (send(sockfd, &file_size_bytes, sizeof(file_size_bytes), 0) == -1)
    {
        perror("Error sending file size");
        fclose(file);
        return -1;
    }

    while (!feof(file))
    {
        size_t bytes_read = fread(buffer, 1, sizeof(buffer), file);
        if (send(sockfd, buffer, bytes_read, 0) == -1)
        {
            perror("Error sending file data");
            fclose(file);
            return -1;
        }
        bzero(buffer, BUFFER_SIZE);
    }
    fclose(file);
    return 0;
}

void *submit(void *args)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1)
    {
        return (void *)-1;
    }

    int tries = 0;
```

```
67      while (tries < MAX_TRIES)
68      {
69          if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == 0)
70              break;
71          sleep(1);
72          tries += 1;
73      }
74      if (tries == MAX_TRIES)
75      {
76          return (void *)-1;
77      }
78
79      if (send_file(sockfd, file_path) != 0)
80      {
81          close(sockfd);
82          return (void *)-1;
83      };
84
85      size_t bytes_read;
86      char buffer[BUFFER_SIZE];
87      while (true)
88      {
89          bytes_read = recv(sockfd, buffer, BUFFER_SIZE, 0);
90          if (bytes_read <= 0)
91              break;
92          // write(STDOUT_FILENO, buffer, bytes_read);
93          bzero(buffer, BUFFER_SIZE);
94      }
95
96      close(sockfd);
97
98      return (void *)0;
99  }
```

```
100
101 int main(int argc, char *argv[])
102 {
103     if (argc != 6)
104     {
105         cerr << "Usage: ./submit  <serverIP:port>  <sourceCodeFileTobeGraded> <lo
    <sleepTimeSeconds> <timeoutSeconds>\n";
106         return -1;
107     }
108
109     string server_ip;
110     int server_port;
111     string ip_port = string(argv[1]);
112     int colon_pos = ip_port.find(':');
113
114     if (colon_pos != std::string::npos)
115     {
116         server_ip = ip_port.substr(0, colon_pos);
117         server_port = stoi(ip_port.substr(colon_pos + 1));
118     }
119     else
120     {
121         cerr << "Invalid input format\n";
122         return -1;
123     }
124
125     file_path = argv[2];
126
127     bzero((char *)&serv_addr, sizeof(serv_addr));
128     serv_addr.sin_family = AF_INET;
129     serv_addr.sin_port = htons(server_port);
130     inet_pton(AF_INET, server_ip.c_str(), &serv_addr.sin_addr.s_addr);
131
```

```
131
132     int loop_num = atoi(argv[3]);
133     int sleep_time_seconds = atoi(argv[4]);
134     int timeout_seconds = atoi(argv[5]);
135     int successes = 0;
136     int total_response_time = 0;
137
138     auto start_loop = chrono::high_resolution_clock::now();
139     for (int i = 0; i < loop_num; i++)
140     {
141         struct timespec ts;
142         pthread_t worker;
143
144         if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
145             continue;
146         ts.tv_sec += timeout_seconds;
147
148         auto start_req = chrono::high_resolution_clock::now();
149         pthread_create(&worker, NULL, submit, NULL);
150
151         if (pthread_timedjoin_np(worker, NULL, &ts) != 0)
152         {
153             pthread_detach(worker);
154             continue;
155         }
156         else
157         {
158             auto end_req = chrono::high_resolution_clock::now();
159             cout << "SUCCESS" << endl;
160             successes++;
161             total_response_time +=
    chrono::duration_cast<std::chrono::microseconds>(end_req - start_req).count();
162         }
163
```

```
163
164         usleep(1000000 * (sleep_time_seconds + rand() * 1.0 / RAND_MAX));
165     }
166     auto end_loop = chrono::high_resolution_clock::now();
167
168     int loop_time = chrono::duration_cast<std::chrono::microseconds>(end_loop -
    start_loop).count();
169
170     cout << "Average response time (seconds): " << (total_response_time * 1.0 /
    successes / 1000000) << "\n";
171     cout << "Total successful responses: " << successes << "\n";
172     cout << "Total time for loop (seconds): " << loop_time * 1.0 / 1000000 << "\n";
173
174     return 0;
175 }
```

Server

```cpp
#include <iostream>
#include <fstream>
#include <cstring>
#include <thread>
#include <filesystem>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <unistd.h>
#include <netinet/in.h>

using namespace std;
namespace fs = std::filesystem;

const int BUFFER_SIZE = 1024;
const int MAX_FILE_SIZE_BYTES = 4;
const int MAX_QUEUE = 50;

const char SUBMISSIONS_DIR[] = "./submissions/";
const char EXECUTABLES_DIR[] = "./executables/";
const char OUTPUTS_DIR[] = "./outputs/";
const char COMPILER_ERROR_DIR[] = "./compiler_error/";
const char RUNTIME_ERROR_DIR[] = "./runtime_error/";
const char EXPECTED_OUTPUT[] = "./expected/output.txt";

const char PASS_MSG[] = "PASS\n";
const char COMPILER_ERROR_MSG[] = "COMPILER ERROR\n";
const char RUNTIME_ERROR_MSG[] = "RUNTIME ERROR\n";
const char OUTPUT_ERROR_MSG[] = "OUTPUT ERROR\n";

```

```cpp

mutex thread_mutex;
int total_requests = 0;
int served_requests = 0;

int recv_file(int sockfd, string file_path)
{
    char buffer[BUFFER_SIZE];
    bzero(buffer, BUFFER_SIZE);
    FILE *file = fopen(file_path.c_str(), "wb");
    if (!file)
    {
        perror("Error opening file");
        return -1;
    }

    char file_size_bytes[MAX_FILE_SIZE_BYTES];
    if (recv(sockfd, file_size_bytes, sizeof(file_size_bytes), 0) ==
    {
        perror("Error receiving file size");
        fclose(file);
        return -1;
    }
    int file_size;
    memcpy(&file_size, file_size_bytes, sizeof(file_size_bytes));

    size_t bytes_read = 0;
```

```
    while (true)
    {
        size_t bytes_recvd = recv(sockfd, buffer, BUFFER_SIZE, 0);
        bytes_read += bytes_recvd;
        if (bytes_read <= 0)
        {
            perror("Error receiving file data");
            fclose(file);
            return -1;
        }
        fwrite(buffer, 1, bytes_recvd, file);
        bzero(buffer, BUFFER_SIZE);
        if (bytes_read >= file_size)
            break;
    }
    fclose(file);
    return 0;
}
```

```
int start_worker(int worker_id, int sockfd)
{
    string source_file = SUBMISSIONS_DIR + to_string(worker_id) + ".cpp";
    string executable = EXECUTABLES_DIR + to_string(worker_id) + ".o";
    string output_file = OUTPUTS_DIR + to_string(worker_id) + ".txt";
    string compiler_error_file = COMPILER_ERROR_DIR + to_string(worker_id
    string runtime_error_file = RUNTIME_ERROR_DIR + to_string(worker_id)

    string compile_command = "g++ " + source_file + " -o " + executable +
2> " + compiler_error_file;
    string run_command = executable + " > " + output_file + " 2> " + runt
    string compare_command = "diff " + output_file + " " + EXPECTED_OUTPU
null 2> /dev/null";

    string result_msg = "";
    string result_details = "";

    if (recv_file(sockfd, source_file) != 0)
    {
        close(sockfd);
        lock_guard<mutex> lock(thread_mutex);
        served_requests++;
        return 0;
    }
```

```
    if (system(compile_command.c_str()) != 0)
    {
        result_msg = COMPILER_ERROR_MSG;
        result_details = compiler_error_file;
    }
    else if (system(run_command.c_str()) != 0)
    {
        result_msg = RUNTIME_ERROR_MSG;
        result_details = runtime_error_file;
    }
    else if (system(compare_command.c_str()) != 0)
    {
        result_msg = OUTPUT_ERROR_MSG;
        result_details = output_file;
    }
    else
    {
        result_msg = PASS_MSG;
    }

    if (send(sockfd, result_msg.c_str(), strlen(result_msg.c_str()), 0) == -1)
    {
        perror("Error sending result message");
        close(sockfd);
        lock_guard<mutex> lock(thread_mutex);
        served_requests++;
        return 0;
    }
```

```
129     if (!result_details.empty())
130     {
131         char buffer[BUFFER_SIZE];
132         bzero(buffer, BUFFER_SIZE);
133 
134         FILE *file = fopen(result_details.c_str(), "rb");
135         while (!feof(file))
136         {
137             size_t bytes_read = fread(buffer, 1, sizeof(buffer), file);
138             if (send(sockfd, buffer, bytes_read, 0) == -1)
139             {
140                 perror("Error sending result details");
141                 fclose(file);
142             }
143             bzero(buffer, BUFFER_SIZE);
144         }
145         fclose(file);
146     }
147     close(sockfd);
148     lock_guard<mutex> lock(thread_mutex);
149     served_requests++;
150     return 0;
151 }
```

```cpp
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        cerr << "Usage: ./server  <port>\n";
        return -1;
    }

    int port = stoi(argv[1]);

    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1)
    {
        perror("Socket creation failed");
        return 1;
    }
    struct sockaddr_in serv_addr;
    bzero((char *)&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);
    int iSetOption = 1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char *)&iSetOption,
sizeof(iSetOption));
    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) != 0)
    {
        perror("Bind failed");
        close(sockfd);
        return -1;
    }
    if (listen(sockfd, MAX_QUEUE) != 0)
    {
        perror("Listen failed");
        close(sockfd);
        return -1;
    }
```

```
    cout << "Server listening on port: " << port << "\n";

    try
    {
        if (!fs::exists(SUBMISSIONS_DIR))
            fs::create_directory(SUBMISSIONS_DIR);
        if (!fs::exists(EXECUTABLES_DIR))
            fs::create_directory(EXECUTABLES_DIR);
        if (!fs::exists(OUTPUTS_DIR))
            fs::create_directory(OUTPUTS_DIR);
        if (!fs::exists(COMPILER_ERROR_DIR))
            fs::create_directory(COMPILER_ERROR_DIR);
        if (!fs::exists(RUNTIME_ERROR_DIR))
            fs::create_directories(RUNTIME_ERROR_DIR);
    }
    catch (fs::filesystem_error &e)
    {
        cerr << "Error creating directories: " << e.what() << "\n";
        close(sockfd);
        return -1;
    }

    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);

    while (true)
    {
        int client_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);

        lock_guard<mutex> lock(thread_mutex);
        thread worker(start_worker, total_requests++, client_sockfd);
        worker.detach();
    }

    close(sockfd);
    return 0;
}
```

C++ Tab Width: 2 Ln 185, Col 6 INS

Load testing script

```bash
#!/bin/bash

make load

SERVER_IP=$(nslookup "$1" | awk '/^Address: / { print $2; exit }')
if [ -n "$SERVER_IP" ]; then
    echo "Starting simulation with $1"
else
    echo "Unable to resolve the IP address of $1"
    exit 1
fi

SERVER_PORT=$2

SOURCE_FILE="./source_code/pass.cpp"

mkdir -p simulation_results

CONCURRENT_CLIENTS=$3
REQUESTS_PER_CLIENT=$4
SLEEP_TIME_SECONDS=$5
TIMEOUT_SECONDS=$6

for ((i=1; i<=$CONCURRENT_CLIENTS; i++)); do
    (
        ./submit "$SERVER_IP:$SERVER_PORT" "$SOURCE_FILE" "$REQUESTS_PER_CLIENT" "$SLEEP_TIME_SECONDS" "$TIMEOUT_SECONDS" > "simulation_results/$i.txt"
    ) &
done

for ((i=1; i<=$CONCURRENT_CLIENTS; i++)); do
    (
        cat "simulation_results/$i.txt"
    )
done

for job in `jobs -p`
do
    wait $job
done
```