Distributed Processing via Networking and Multithreading in C++:
An Image Processing Application
CMP9133M

Stewart Charles Fisher II
School of Computer Science
University of Lincoln
Lincoln, United Kingdom
25020928@students.lincoln.ac.uk

***Abstract*—This report is a description and explanation of a server-client application, used to submit an image and apply graphical filters before returning a modified image to the user, showcasing the usage of networking and multithreading in C++. The report will justify the choices made, address key challenges and solutions, and provide an outline on the future-proof quality of the application.**

***Index Terms*—Image processing, client-server architecture, multithreading, multiprocessing, visualisation**

I. System Design

A. Architecture Overview

The server-client architecture is primarily derived from a base class, `Peer`, which provides the fundamental image transmission functions and data fragmentation size, while the two derived classes, `Client` and `Server`, provide the task-specific functionalities. Both the server and client classes utilise functions held within helper constructs. The server uses the `ThreadPool` class and the `ImageFilter` class while the client uses the `FilterRequirement` structure and `ParamType` enumeration. The UML[1] class diagram displaying the server-client structure can be seen in Appendix A. The UML sequence diagram outlining example of a successful client-server operation can be seen in Appendix C.

B. The `Client` Class

The `Client` class validates user inputs for filter operations and parameters, ensuring robust interaction. The class manages sending image data and processing instructions to the server. Additionally, it incorporates error handling and network communication protocols, emphasising reliability and efficiency in client-server interaction.

C. The `Server` Class

The `Server` class handles multiple concurrent client connections using a thread pool, provided by the `ThreadPool` class. The class receives, processes, and returns images after applying a user-specified filter, which are provided by the `ImageFilter` class family. The usage of thread pooling ensures that the processing of the images is more resource and time efficient. The UML class diagram displaying the class family can be seen in Appendix B.

D. The `ImageFilter` Class Family

The `ImageFilter` class family is used to define a suite of modular image filter classes, with each of them being specialised in a type of image manipulation. The OpenCV library is used to apply processing for the image manipulation. A total of 14 filters are included in this implementation.

E. The `ThreadPool` Class

The `ThreadPool` class uses parallel execution and concurrency of tasks using multiple worker threads, with a synchronised queue using mutexes and condition variables are used for thread-safe operations. Modern solutions have been employed, such as the usage of `std::future` and `std::packaged_task` to allow tasks to return results asynchronously.

F. The `ParamType` Enumeration

The `ParamType` enumeration is used to ensure the given parameter is of the correct type, to ensure runtime type safety.

G. The `FilterRequirement` Structure

The `FilterRequirement` structure is used to ensure that a given operation is allowed, and to also check that the given parameter is allowed, using the `ParamType` enumeration to validate the parameter type.

II. Implementation

A. Data Transmission

TCP was chosen to transmit the images and the parameters. Even though the UDP protocol allows for faster, real-time transmission, it is not best suited for suitations that require reliable and safe transmissions, such as file transfers. Furthermore, using TCP affords me to use a more robust implementation when compared to a possible custom implementation.

[1]The UML diagrams were generated using PlantUML.

Sending the entire image in a single transmission caused the image to be received in a corrupted state. In order to solve this, I implemented a system that loops, sending and receiving a predetermined fragment size, until an end-of-transmission character is received.

While the server application is set to using the default localhost address, the user is able to specify the target address for the client application. Rigorous error management has been provided to ensure the user is aware of any faults, such as the inability to secure socket connections.

B. Concurrency

The thread pool provided by the `ThreadPool` class allows the application to sustain simultaneous client requests concurrently, thereby ensuring efficient but isolated processing of each client's specific task. In this application, the default number of worker threads employed is 4 threads, however that is able to be changed within the `server.cpp` source file.

The thread usage could be improved by using `std::jthread`, which automatically handles cleanup on destruction, allowing for better memory management and safety.

C. Image Filters

The image filters are arranged in a class hierarchy stemming from a base class, `ImageFilter`, which affords a large amount of modality in filter operation if necessary at a later point. This hierarchy heavily exhibits the usage of object-oriented programming techniques, such as inheritance, polymorphism, access modification, and abstraction.

Abstract classes such as `ColourAdjustFilter` are used to collect similar filter classes, aiding the future-proof quality of the code. If a change needs to be made at a later point that should affect all derived classes, such as the introduction of a new member variable or functions, this method preemptively ensures that code doesn't have to be redundantly reproduced across the relevant classes.

D. Memory Management

The application relies heavily on the usage of references and pointers to reduce the unnecessary copying of data. An example is how the `_createFilter_` function uses the modern standard of dynamic memory allocation through smart pointers, which automatically deallocate memory once the object is out of scope; this can be seen in Appendix D. Furthermore, mutexes are used for shared resources to manage concurrent access, preventing race conditions. Prevention on race conditions is crucial in multithreaded application like this.

E. Code Practice

- Private variables are clearly marked using underscore prefixing and suffixing to distinguish from parameter variables that share the same name.
- Header files are used to improve the modularity and scalability of the system, while reducing the code redundancy.

F. Lambda Expressions

Lambda expressions are used to efficiently create tasks for the thread pool, capturing the context, `this`, and avoiding the need to define a separate function to pass to enqueue. Using this, the code remains readable and concise. This can be seen in Appendix E.

G. Template Files

A template file, `threadPool.tpp`, is used by the `ThreadPool` class to contain the manual implementation of the `enqueue` template function; this can be seen in Appendix F. The function template takes a function and its arguments and wraps them to create an executable task, which is then added to the queue. Once a task is queued, a condition variable is notified to signal a thread in the pool to execute that task, returning a `future` object that allows the results to be obtained at completion.

Using template files keeps the header file clean and readable. The template file is included at the end of the header file so that the necessary files can still access it.

H. Multi-Platform Compatibility

This system was developed entirely using a Linux system. POSIX-compliant systems use a different socketing API to Windows, which uses the Winsock API. In order to support multi-platform compatibility with Windows systems, preprocessing directives i.e. `#ifdef _WIN32`, are used to include platform-specific socket code. Furthermore, the platform-specific code is used to initialise and cleanup Winsock, using `WSAStartup` and `WSACleanup` respectively. Despite this, it should be noted that the application has only been tested on a Linux system and not on a Windows system, thereby meaning there could potentially issues with the compilation and execution.

Due to the number of files included in this implementation, CMake was used to organise and contain the compilation. The CMake configuration file can be seen in Appendix G.

III. Scalability

As the number of client requests increases, the number of threads might become insufficient to handle tasks efficiently, leading to an increase in response times. In order to handle this, the number of threads should be manually increasing the number of threads, or implementing a dynamic thread pool that adjusts according to workload. Furthermore, the number of threads available on the hardware could demand an upgrade to server-grade hardware to have access to more threads.

Switching to using UDP from TCP could also address the challenges of a large-scale operation, allowing for lower overhead and faster data transmission. As such, real-time processing would be more beneficial as more clients try to access the server. A hybrid approach could also be explored, using both protocols, where the image is transferred using TCP, to ensure correct transmission, whilst the string inputs for the operation and parameter are communicated via UDP.

APPENDIX A
SERVER-CLIENT INFRASTRUCTURE DIAGRAM

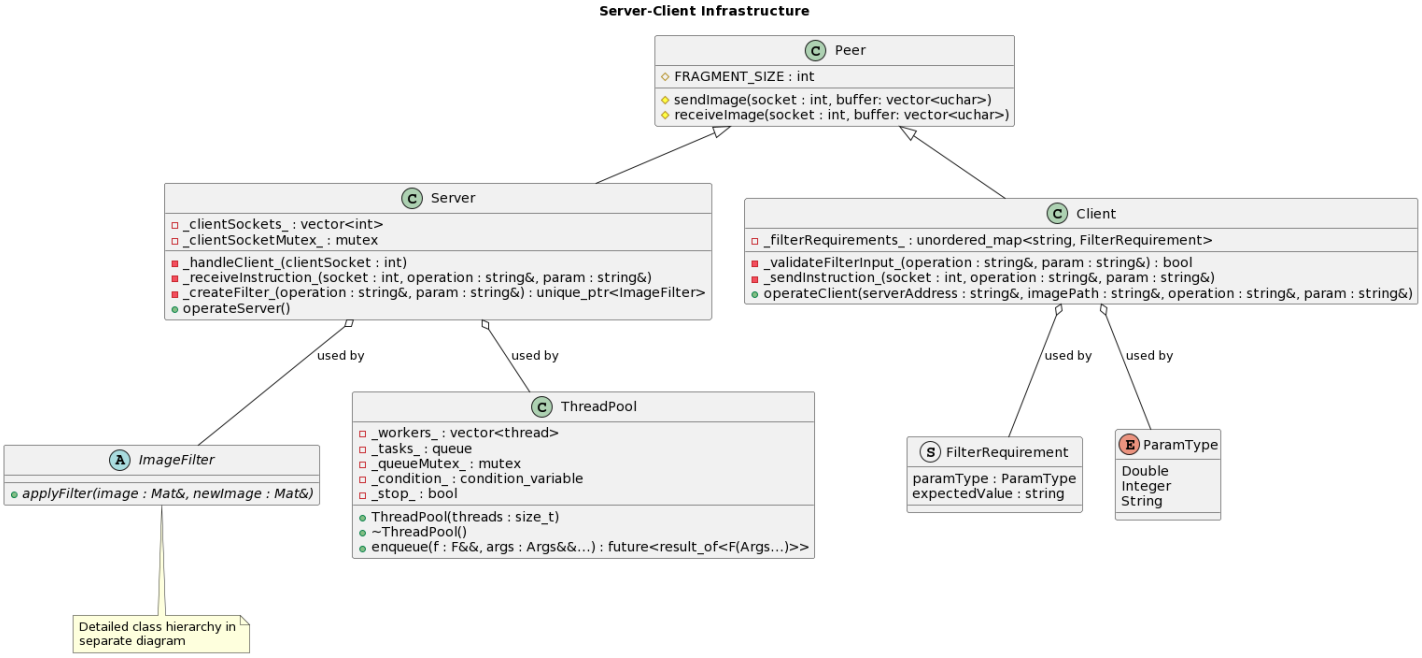

Fig. 1. A UML class diagram for the server-client infrastructure.

APPENDIX B
IMAGE FILTER CLASS HIERARCHY DIAGRAM

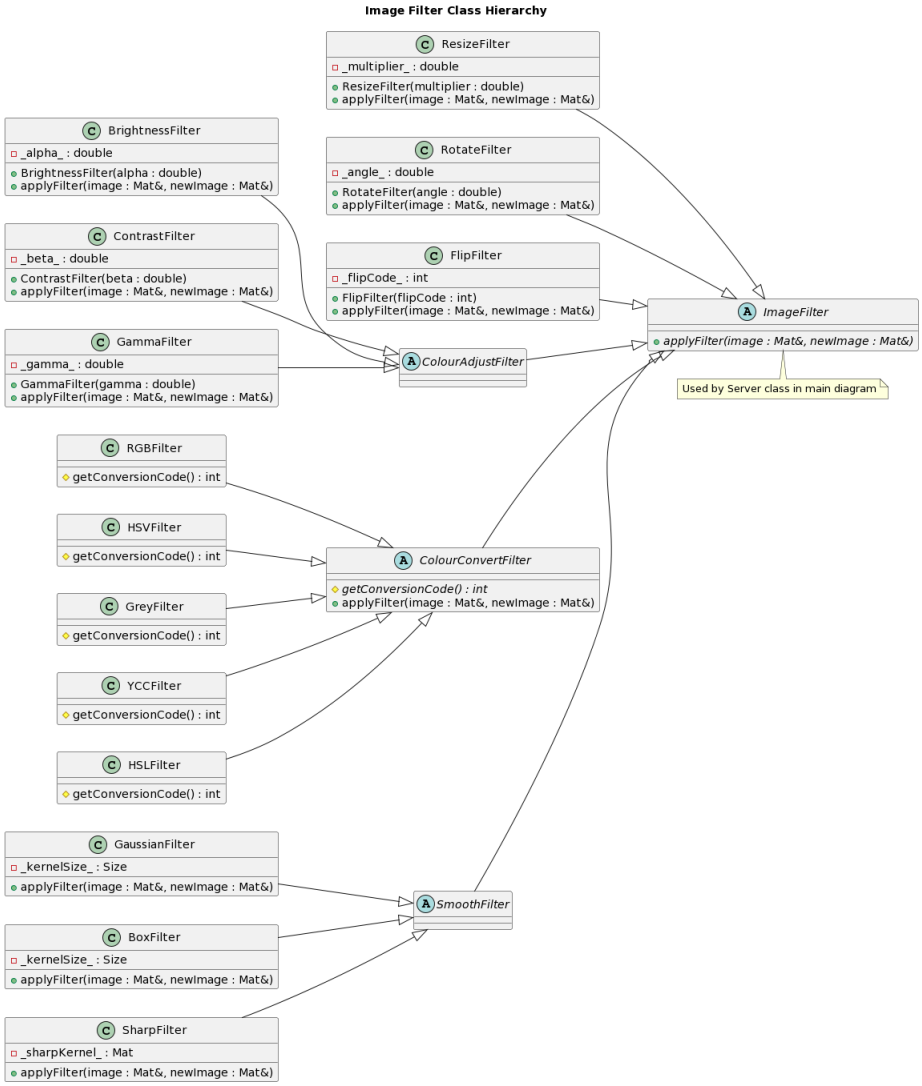

Fig. 2. A UML class diagram for the image filter hierarchy.

APPENDIX C
SERVER-CLIENT INTERACTION SEQUENCE DIAGRAM

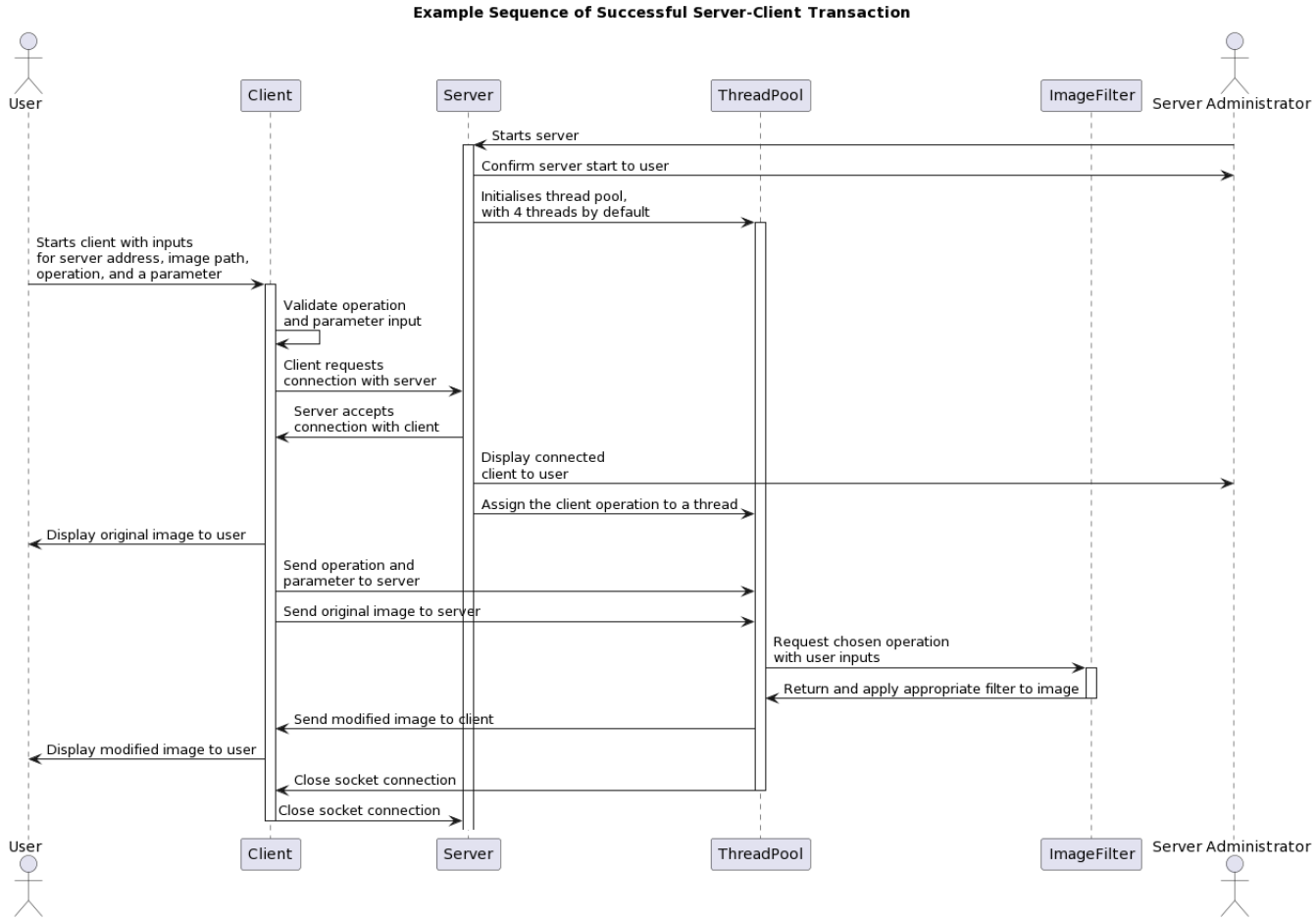

Fig. 3. A UML sequence diagram for the server-client interaction.

APPENDIX D
SMART POINTER FUNCTION

```
std::unique_ptr<ImageFilter> Server::_createFilter_(
    const std::string& operation, const std::string& param) {
  std::istringstream iss(param);
  double numericParam;
  int intParam;

  // Return the appropriate filter
  if (operation == "resize") {
    iss >> numericParam;
    return std::make_unique<ResizeFilter>(numericParam);
...
  // Handle unusable cases
  return nullptr;
}
```

Listing 1. A code snippet from the `_createFilter_` function.

APPENDIX E
LAMBDA FUNCTION

```
void Server::operateServer() {
...
     // Create a thread for the new client using lambda function
     pool.enqueue(
        [this, clientSocket]() { this->_handleClient_(clientSocket); });
...
}
```

Listing 2. A code snippet of lambda function usage.

APPENDIX F
THREAD POOL ENQUEUE FUNCTION

```
// Copyright 2023 Stewart Charles Fisher II

// ThreadPool enqueue template
template <class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type> {
    using return_type = typename std::result_of<F(Args...)>::type;

    // Wrap the function and its arguments into a task
    auto task = std::make_shared<std::packaged_task<return_type()>>(
        std::bind(std::forward<F>(f), std::forward<Args>(args)...));

    // Get the future for the task's result
    std::future<return_type> res = task->get_future();
    {
        // Lock the mutex
        std::unique_lock<std::mutex> lock(_queueMutex_);

        // If the pool has stopped, throw an exception
        if (_stop_) throw std::runtime_error("enqueue on stopped ThreadPool");

        // Add the task to the queue
        _tasks_.emplace([task]() { (*task)(); });
    }
    // Notify a waiting thread
    _condition_.notify_one();

    // Return the future
```

```
    return res;
}
```

Listing 3. The template file, `threadPool.tpp`.

APPENDIX G
CMAKE CONFIGURATION FILE

```
# CMakeLists.txt

cmake_minimum_required(VERSION 3.12)
project(DistributedProcessing)

# Export compile commands
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# OpenCV
find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})

# Set the src directory
set(SRC_DIR ${CMAKE_CURRENT_SOURCE_DIR}/src)

# Server executable
add_executable(server ${SRC_DIR}/server.cpp ${SRC_DIR}/processing.cpp ${SRC_DIR}/processing.h $
    {SRC_DIR}/peer.cpp ${SRC_DIR}/peer.h ${SRC_DIR}/threadPool.cpp ${SRC_DIR}/threadPool.h)
target_link_libraries(server PRIVATE ${OpenCV_LIBS})

# Client executable
add_executable(client ${SRC_DIR}/client.cpp ${SRC_DIR}/peer.cpp ${SRC_DIR}/peer.h)
target_link_libraries(client PRIVATE ${OpenCV_LIBS})

# Windows-specific compilation
if(WIN32)
  target_link_libraries(server PRIVATE Ws2_32)
  target_link_libraries(client PRIVATE Ws2_32)
endif()

# Set the output directory for the executables
set_target_properties(server PROPERTIES RUNTIME_OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR
    }/../bin")
set_target_properties(client PROPERTIES RUNTIME_OUTPUT_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR
    }/../bin")
```

Listing 4. The CMake configuration file to compile the executables.