

# CS6675 Final Project:

## A Linux High-Performance Web Server

Hongnan Zhang

<b>1 Introduction .....</b>	<b>- 2 -</b>
1.1 Background .....	- 2 -
1.2 Existing Web Server Application .....	- 2 -
1.3 Project Plan .....	- 2 -
<b>2 Need-finding Analysis and Preliminary Evaluation .....</b>	<b>- 3 -</b>
2.1 Heuristic-Based Evaluation .....	- 3 -
2.2 Preliminary Empirical Measurements .....	- 5 -
<b>3 Baseline Design and Measurement Results .....</b>	<b>- 6 -</b>
3.1 Procedural Sketch .....	- 6 -
3.2 Architecture .....	- 8 -
3.3 Prototype Implementation .....	- 10 -
<b>4 Redesign through Refinements .....</b>	<b>- 12 -</b>
4.1 Connection Timer .....	- 12 -
4.2 Proactor Pattern .....	- 13 -
4.3 Borderline Cases .....	- 13 -
4.4 Implementation .....	- 14 -
<b>5 Empirical Evaluation Plan .....</b>	<b>- 15 -</b>
5.1 Stress Test .....	- 15 -
5.2 Evaluation Metrics .....	- 16 -
<b>6 Evaluation Execution and Results .....</b>	<b>- 17 -</b>
6.1 Baseline Prototype Result .....	- 17 -
6.2 Refined Prototype Result .....	- 17 -
6.3 Analysis .....	- 17 -
6.4 API Design .....	- 18 -
<b>7 Concluding Remarks .....</b>	<b>- 20 -</b>
<b>Appendix .....</b>	<b>- 21 -</b>
<b>Reference .....</b>	<b>- 24 -</b>

# 1 Introduction

## 1.1 Background

The rapid growth of the Internet has led to a surge in web applications and services, with an increased demand for web servers. As more businesses and individuals rely on web services, it is crucial to ensure that web servers can handle a large number of concurrent connections without compromising performance. Extending from my previous assignment M4: A Web Server Architecture for High Concurrency, the proposed task for this final project is to design and implement a high-performance web server based on Linux kernel, which supports high concurrent TCP connections.

The importance of high-performance web servers lies in their ability to ensure the timely delivery of web contents to users. As the internet and web applications grown, the demand for high-performance web servers has increased to meet the needs of users who require fast and reliable access to web contents. High performance web servers can handle a significant volume of requests simultaneously, which is the key in providing an pleasure user experience while ensuring high availability.

The target users of high-performance web server may be enterprises and organizations that require high quality web services, such as e-commerce sites, online gaming platforms, social media networks, and cloud-based applications. These users require a web server that can handle large volumes of traffic, deliver content quickly, and can provide high levels of reliability.

## 1.2 Existing Web Server Application

There are many existing web servers available in the industry, such as Apache and Nginx, they are widely used for web service hosting and content delivery. Takings the famous open source web server architecture Nginx as an example, it employs an event-driven, asynchronous architecture that enables it to handle a large number of concurrent connections with minimal resource utilization ([Soni, R 2016](#)). This architectural design is both efficient and scalable, allowing Nginx to maintain optimal performance under heavy loads.

Furthermore, Nginx uses asynchronous and non-blocking I/O operations to handle incoming requests. These operations enable the server to process multiple requests simultaneously and minimize waiting time for I/O intensive scenarios.

## 1.3 Project Plan

Honestly, developing a web server that can comprehensively surpass the performance of well-configured solutions like Nginx, which has been thorough extensive testing, optimization, and has a large user community contributing to its development, is undoubtedly challenging. Consequently, this project will

concentrate on extending and validating existing server architecture and resource management technologies. These server optimization technologies will be implemented based on the Linux kernel, followed by an evaluation of performance using stress testing tool.

With personal interest in server-end programming, I conducted a study on two effective concurrent event handling patterns, the Reactor pattern and the Proactor pattern in Assignment M4. Building upon this research, I plan to implement these two patterns within a Linux web server for verification and evaluation of the previously explored concepts.

## 2 Need-finding Analysis and Preliminary Evaluation

### 2.1 Heuristic-Based Evaluation

In order to help prioritize problem areas and facilitates iterative design improvements, a heuristic-based need finding evaluation on the main components is conducted in this section.

The main components of the proposed high performance web server includes: I/O multiplexing mechanism, concurrent event handling pattern, and pooling strategy.

#### 2.1.1 I/O Multiplexing Mechanism

I/O multiplexing is a technique used in computer systems, including web servers, to efficiently handle multiple I/O streams concurrently with a single process or thread. It allows a server to monitor and manage multiple sockets without the need to create separate threads or processes for each one ([Stevens, W. R. 1990](#)).

##### Pros:

- Non-blocking I/O: By using non-blocking I/O calls, applications can continue processing other tasks while waiting for I/O operations to complete, enhancing overall performance and responsiveness.
- Resource efficiency: I/O multiplexing enables a single process or thread to handle multiple I/O streams, which can lead to more efficient resource usage compared to a multi-process or multi-threaded model.

##### Con:

- Platform-dependent: Some I/O multiplexing APIs, such as epoll and kqueue in Linux, are platform-specific, which may limit the migration capacity of the web server across different operating systems.

#### 2.1.2 Concurrent Event Handling Pattern

Concurrent event handling patterns, like Reactor and Proactor mentioned in

M4, are event-driven architectural pattern for handling multiple I/O events in a non-blocking, synchronous or asynchronous manner. These patterns provide a structured approach to managing the concurrency, ensuring efficient processing of client requests and optimal use of server resources ([Kerrisk, M. 2012](#)).

**Pros:**

- Improved responsiveness: Concurrent event handling can help applications remain responsive, as multiple events can be processed simultaneously without blocking other tasks.
- Resource efficiency: Both Reactor and Proactor patterns can dispatch multiple events with a single thread, reducing resource overhead while improving efficiency.
- Scalability: By efficiently managing concurrency, event handling patterns allow applications to scale with increasing loads and connection numbers.

**Con:**

- Suitability: Choosing the right pattern for a specific use case can be challenging, as the effectiveness of the pattern depends on the nature of the tasks and I/O operations involved.

### 2.1.3 Pooling Strategy

Pooling is a collection of established connections or pre-allocated worker threads that are used to process incoming client connections and requests. Frequently creating and destroying threads or connections can introduce unnecessary overheads in terms of system resources. By using pooling strategies, web servers can reduce the overhead associated with these operations, resulting in more efficient processing of client requests and better overall performance.

**Pros:**

- Resource efficiency: Pooling helps optimize resource usage by reusing resources and preventing the creation of excessive threads or connections, resulting in lower resource consumption and better overall performance.
- Simplified resource management: Pooling provides a structured approach to managing resources, making it easier to monitor and control resource usage within the web server.

**Cons:**

- Potential for resource under-utilization: If the pool size is not configured promptly, it may lead to an under-utilization of resources, with resources sitting idle in the pool without being used.

- Connection pool-specific: Connection pools may hold stale or broken connections, which can lead to errors or delays in processing requests. Regular health checks and monitoring may be required to ensure the quality of connections in the pool (You, S. 2013).

## 2.2 Preliminary Empirical Measurements

To contribute to the baseline design and implementation, a preliminary empirical review will be conducted on the performance of existing web server applications. Besides, it is also essential to establish a baseline measurement plan on conducting performance measurement for the prototype.

### 2.2.1 Empirical Overview of Mainstream Solutions

As mentioned in 1.2, Nginx is renowned for its high-performance, event-driven architecture, and asynchronous I/O, which allows it to handle a large number of concurrent connections with little resource usage, and to maintain high request throughput and low response times even under heavy loads.

Apache, a widely used and highly configurable web server, offers a modular architecture and support for a wide range of extensions. While its traditional process-based or thread-based architecture can handle moderate workloads, it may result in increased resource consumption when managing a high number of concurrent connections (Kunda, D. 2017). This may lead to lower request throughput and increased response times compared with event-driven server Nginx.

Lighttpd is a lightweight web server specifically designed for speed-critical environments. Its similar event-driven architecture provides excellent performance in terms of request throughput and response time, particularly for serving static content. While Lighttpd's smaller feature set and community size may limit its applicability in more complex or demanding web applications, its performance benefits make it a compelling option for specific use cases.

### 2.2.2 Performance Measurement Plan

- Measurement metrics: To effectively measure the performance, the measurement plan should include the system throughput and response time. Throughput can be measured in queries per second (QPS), response time in milliseconds (ms).
- Performance tests: To execute performance tests, it is essential to use stress test tools to simulate heavy workloads and traffic patterns.
- Results analysis: Subsequently, the test results against the objectives will be compared to evaluate the web server's performance, identify any bottlenecks or performance issues, and determine whether additional optimizations or configuration adjustments are required.
- Refinements: Then, make design refinements to the web server's

configuration or optimizations and repeat the testing process.

## 3 Baseline Design and Measurement Results

### 3.1 Procedural Sketch

The procedural sketch for the functionality of the proposed web server, that supports concurrent TCP connection and handles HTTP requests is outlined as follows:

#### 1. Initialize server components:

- Create a listening socket for incoming connections.
- Configure the socket with necessary options, such as non-blocking mode and reuse of the local address.
- Bind the socket to the server's IP address and port number.
- Set the socket to listen for incoming connections.

#### 2. Set up the I/O multiplexing mechanism:

- Create an Linux epoll instance.
- Register the listening socket with the epoll instance, monitoring incoming connections.

#### 3. Initialize the Reactor pattern:

- Implement a main event loop that waits for events triggered by the I/O multiplexing mechanism.
- Define event handlers for different event types, such as new connections, read/write ready events, and closed connections.

#### 4. Set up the thread pool:

- Create a fixed-size thread pool.
- Implement a task queue to store incoming HTTP requests.
- Assign worker threads to process tasks from the queue.

#### 5. Initialize MySQL database and connection pool:

- Set up the MySQL database with appropriate tables and indexes for user authentication.
- Create a database connection pool to manage concurrent database connections efficiently.

#### 6. Main event loop:

- Run the main event loop, waiting for events from the epoll instance.

- Upon receiving a new connection event, accept the connection and register the new socket with the epoll instance.
- When data is received on a socket, parse the HTTP request and add the task to the task queue.
- Worker threads in the thread pool will process tasks from the queue, handling HTTP requests such as user login, requesting web contents.

7. Cleanup and shutdown:

- When the server is shut down, close all active sockets, terminate worker threads, and release resources such as the epoll instance and database connection pool.

### 3.2 Architecture

The architecture of proposed the high performance web server includes several main functional components.

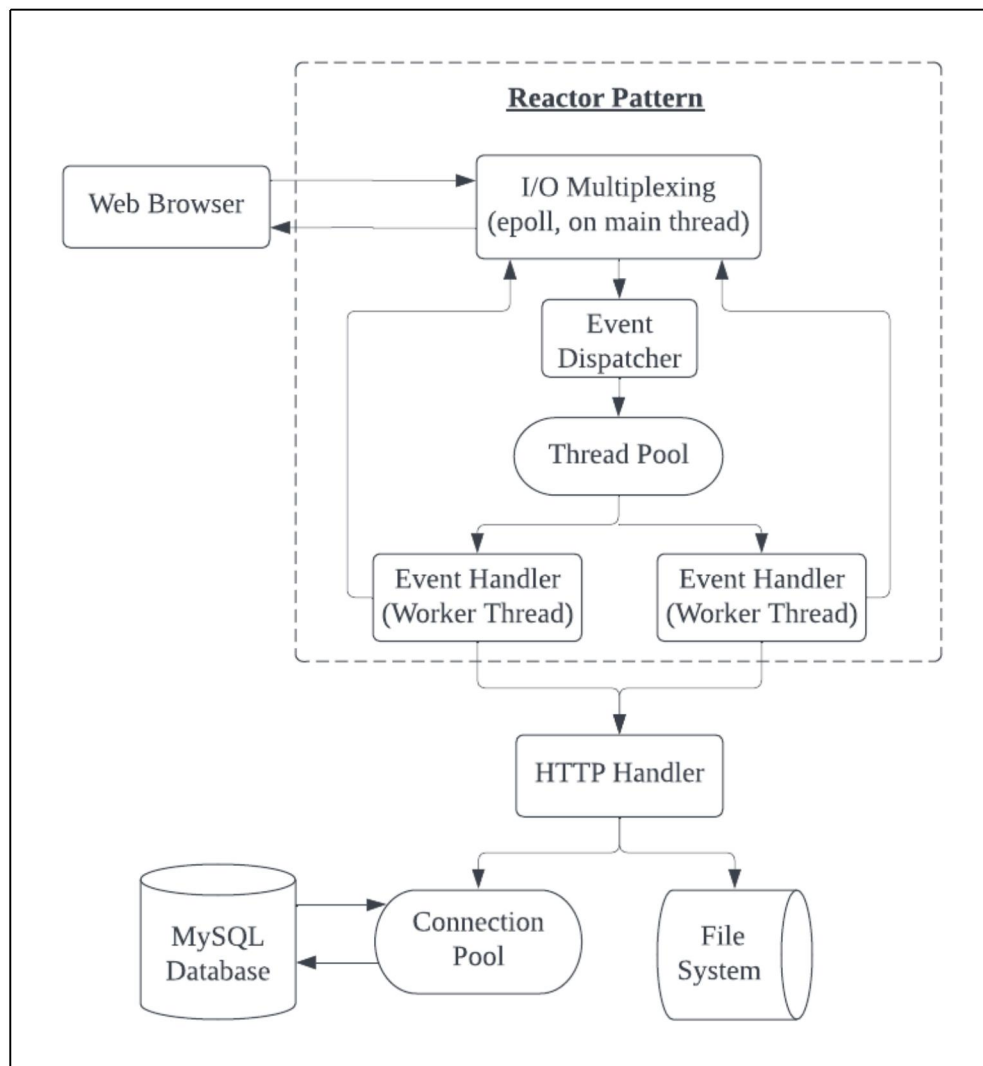


Figure 1. Baseline Design Architecture

- **I/O Multiplexing (epoll):** The Linux **epoll** API is the foundation of the web server's I/O multiplexing mechanism. It enables the server to efficiently handle multiple simultaneous connections without incurring the overhead of one thread per connection. **Epoll** monitors multiple file descriptors to identify when they are ready for I/O operations, thus ensuring efficient utilization of server resources.
- **Reactor Pattern:** This concurrent event handling pattern is built on top of the **epoll** mechanism. In reactor pattern, the main thread manages incoming events, registers them to an event dispatcher, and monitors activities. Upon occurrence, it alerts the server application, invoking a worker thread for processing tasks like socket data transfers, before awaiting the next event ([My Assignment M4, 2023](#)). Detailed explanation



on Reactor pattern please see in [Appendix 1](#).

- *Thread Pool:* A thread pool is utilized to manage multiple worker threads that process incoming HTTP requests. By reusing a fixed number (corresponding to CPU cores) of threads, the web server avoids the overhead of dynamically creating and destroying threads for each incoming request. The worker threads process tasks from a shared task queue, which stores incoming HTTP requests. This approach ensures efficient utilization of server resources and maintains high throughput.
- *HTTP Handler:* The HTTP handler receives raw HTTP requests and parses them into structured data. It involves extracting the request method (GET and POST in the proposed prototype), the requested URL and query parameters. Once the processing is complete, the handler collects the resulting data and assembles it into a formatted HTTP response.
- *MySQL Database and Connection Pool:* A MySQL database is used to store user authentication data, enabling the web server to provide login services for concurrent users. The database connection pool manages multiple concurrent connections to the MySQL database, preventing performance degradation caused by establishing and closing database connections for each incoming request.

### 3.3 Prototype Implementation

The baseline prototype is developed in C++, it consists of three main components: the event handler (main.cpp), the HTTP handler (http\_handler.cpp), the connection pool handler (connection\_pool.cpp), and the thread pool handlers (connection\_pool.h and thread\_pool.h).

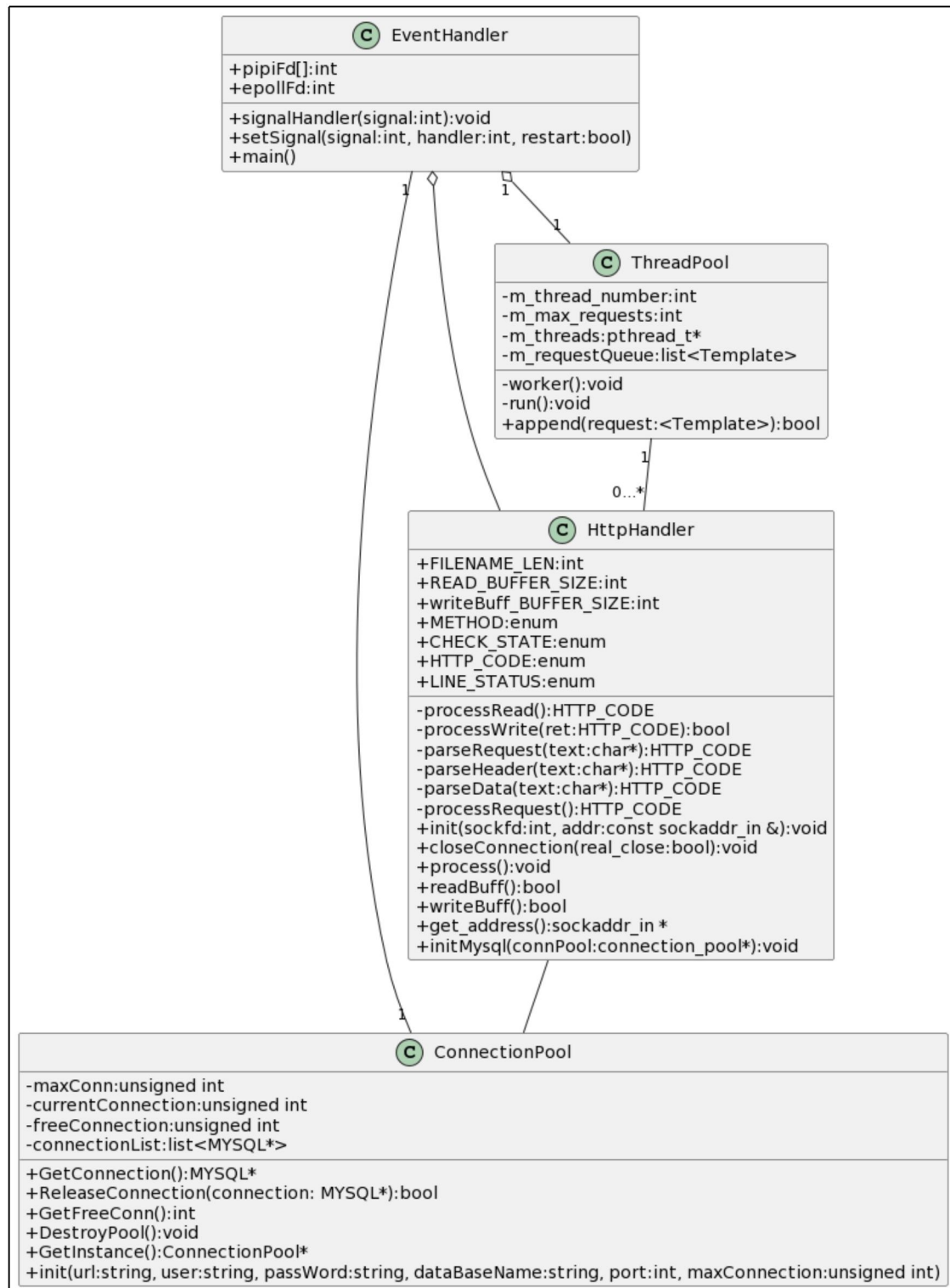


Figure 2. Baseline Prototype Class Diagram

### **3.3.1 Event Handler (Main)**

The event handler performs the following tasks:

- Initializes kernel I/O event table, thread pool, connection pool, logging, timer, listening socket, and signals.
- Handles I/O events using the synchronized reactor pattern, with listening, receiving, and responding to HTTP requests only occurring on the main thread.
- Worker threads in the thread pool are responsible for reading and writing buffers, parsing HTTP messages, and processing requests.
- When a new request arrives, the listening socket registers it with the request queue and event dispatcher. A worker thread is then activated from the thread pool and assigned the request from the queue.
- After processing the request, a write event to server's send buffer is performed by the worker thread.
- The main thread writes the response to the socket and sends it to the client when available.
- For POST requests requiring database access, a MySQL connection is activated from the connection pool and assigned to the worker thread.

### **3.3.2 HTTP Handler**

The "HttpHandler" class performs the following tasks:

- When a client (browser) initiates a new HTTP connection, the main thread creates a new HTTP instance and stores the request in its receive buffer.
- The HTTP instance is then added to a request queue, and a worker thread processes the request carried by the instance.
- Upon acquiring the instance, the worker thread calls "processRead" to read and parse the HTTP message.
- Finally, "processRequest" is called to generate a response, which is written to the send buffer and sent to the client on the main thread.

### **3.3.3 Connection Pool**

The "ConnectionPool" class performs the following tasks:

- Defined utilizing the Singleton pattern.
- The resources in the "ConnectionPool" consist of a set of statically allocated database connections that are dynamically used and released by the process.
- When the process requires database access, it obtains a free connection

from the pool.

- After completing the database operation, the process releases the connection back to the pool.

### 3.3.4 Thread Pool

The “ThreadPool” class performs the following tasks:

- It allocates multiple threads statically during initialization, minimizing the overhead of frequently creating and deleting threads.
- The main thread listens on the server port for new HTTP requests.
- When a new request arrives, it is stored in a request queue (a list).
- A worker thread from the “ThreadPool” is then awakened to acquire and handle the new HTTP request from the request queue.

## 4 Redesign through Refinements

### 4.1 Connection Timer

Efficient connection management like connection timer can reduce the overhead associated with establishing and closing connections repeatedly, leading to improved performance and faster response times for the clients' requests. Meanwhile, it ensure that server resource such as memory, file descriptors and processing power is allocated to active connections and task.

Connection timers are used to monitor and manage connections, ensuring optimal resource usage and server performance. They can detect idle connections and close them, freeing up resources for other active connections. Timer mechanism has various advantages:

- Resource management: Timers help identify and close idle connections, preventing resource exhaustion and ensuring that the server can accommodate new incoming connections.
- Improved performance: By closing inactive connections, timers ensure that the server's resources are directed towards active connections, improving overall performance.
- Protection against slow clients: Timers can detect slow clients that might otherwise consume server resources for extended periods, preventing them from causing performance degradation or denial of service.

## 4.2 Proactor Pattern

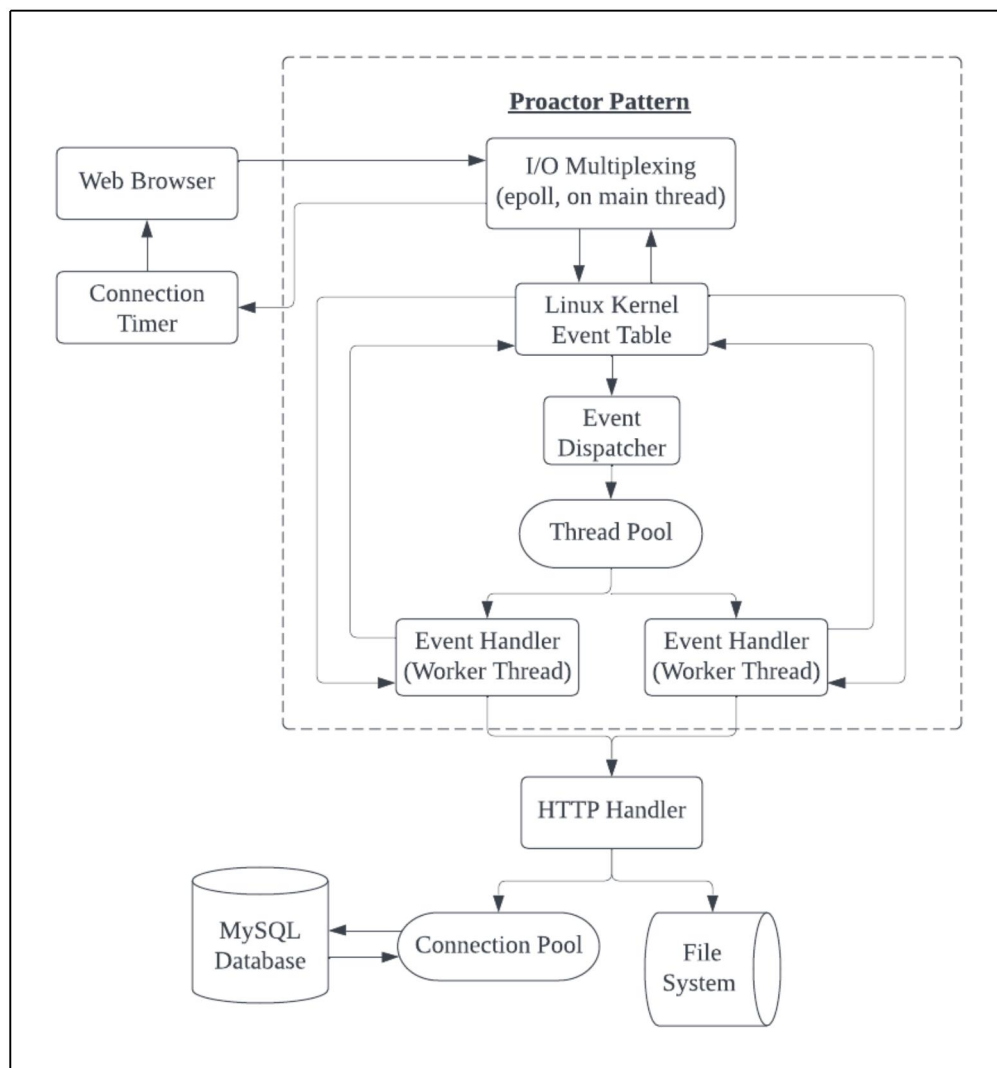


Figure 3. Refined Design Architecture

The Proactor pattern is an asynchronous I/O event-driven design pattern used in high performance web servers. Different from Reactor pattern, it relies on the Linux kernel's asynchronous I/O operations and decouples I/O operations like reading and writing from their completion handlers (worker threads). All I/O operations are performed by kernel rather than worker threads in Reactor pattern, and inter-threads communication uses event-completion signal rather than event-ready signal, in order to utilize efficiency asynchronous operation and avoid potential congestion ([My Assignment M4, 2023](#)). Detailed explanation on Proactor pattern please see in [Appendix 2](#).

## 4.3 Borderline Cases

However, there are borderline cases where Proactor may fail to be effective:

- **Limited asynchronous I/O support:** In systems where the operating system lacks efficient support for asynchronous I/O operations or doesn't support

them at all, the Proactor pattern may not be effective, and the Reactor pattern may be a better choice.

- **Heavy computational tasks:** If the completion handlers require significant computational resources or time, the Proactor pattern's advantages may be reduced, as the server's resources are spent on processing completion handlers rather than handling new connections.

## 4.4 Implementation

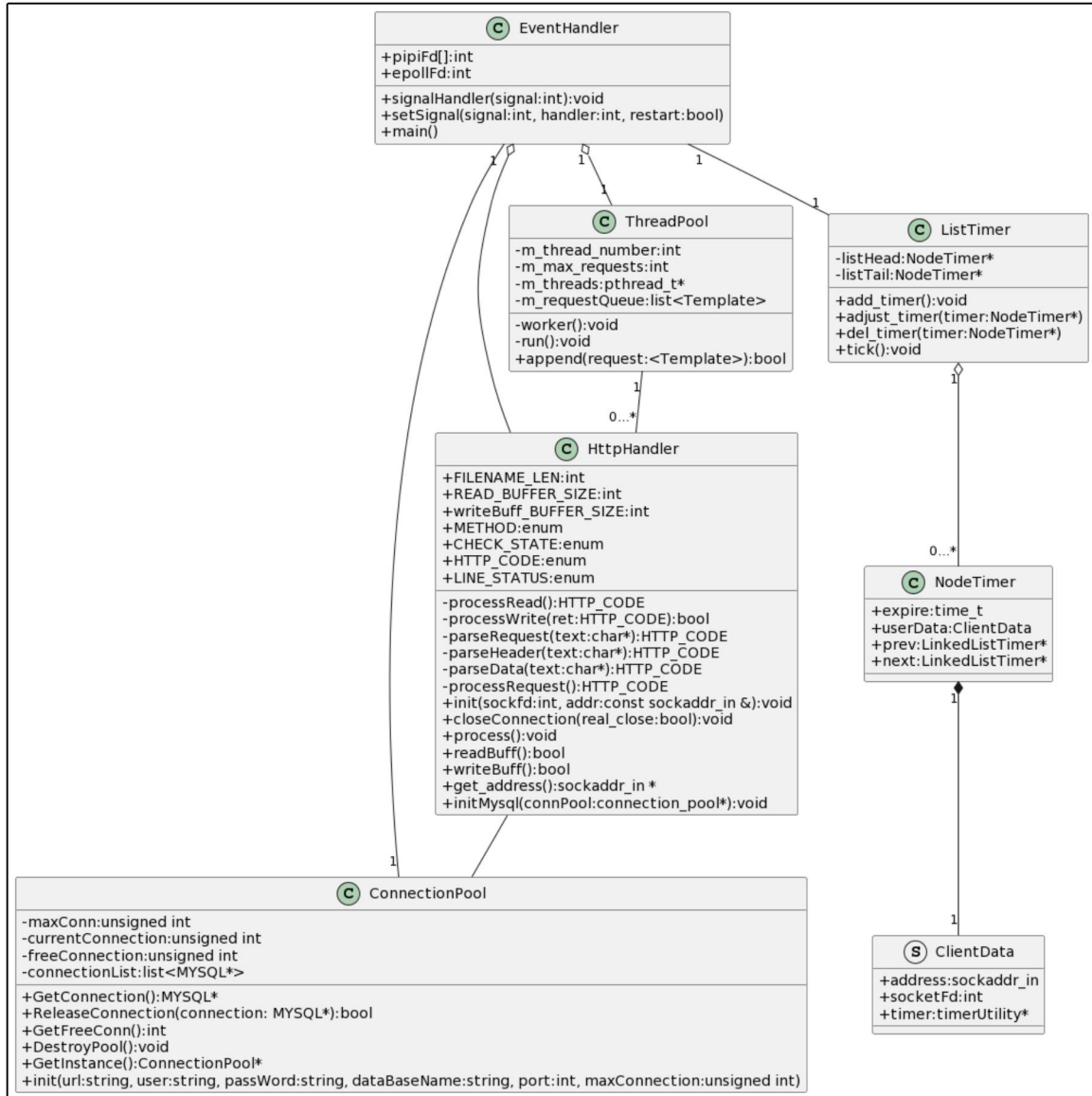


Figure 4. Refined Design Class Diagram

### 4.4.1 Event Handler (Main)

The event handler performs the following tasks:

- Initialize the kernel IO event table, thread pool, connection pool, log, timer, listen socket, and signals.

- IO events are managed using proactor pattern, where listening, receiving, and responding to HTTP requests take place exclusively on the main thread. Other threads in the thread pool act as worker threads, responsible for reading and writing buffers, parsing HTTP messages, and processing requests.
- When a new request reaches the server port on the main thread, the listen socket registers it in the kernel event table and a request queue, and associates it with a timer. A free worker thread from the thread pool is then activated and assigned the request from the queue. Once the request is processed, a write event is registered in the kernel event table.
- Then main thread writes the response to the socket and sends it to the client when available.
- For POST requests that require database access, a free MySQL connection is activated from the connection pool and assigned to the worker thread.
- Each connection is bound to a timer, which the main thread periodically checks for timeout events, triggered by SIGALRM. If a connection remains inactive for a certain period without interacting with the server, the main thread considers it inactive and closes it.

#### **4.4.2 Connection Timer**

The “ListTimer” class performs the following tasks:

- Create a list of timer using an ascending doubly-linked list, where timers are sorted by their expiration time from shortest to longest (head expires first, tail last).
- Each connection's expiration time is 15 seconds from its initialization or last interaction with the server, with any data exchange resetting the timer.
- The root process checks the timer list every 5 seconds, deeming connections that have timed out as inactive and closing them.

## **5 Empirical Evaluation Plan**

### **5.1 Stress Test**

For the proposed web server, a common method of performance evaluation is stress test. It is a testing methodology used to evaluate the stability and reliability of a system under extreme conditions, such as high traffic loads, limited resources, or other constraints. It helps to identify potential bottlenecks, performance issues, and possible failure points. In the context of a web server, a stress test would involve simulating a large number of simultaneous connections and requests to see how the server responds under pressure.

### 5.1.1 Stress Test Tool: Web Bench 1.5

Web Bench 1.5 ([home page](#)) is an open-source web server benchmarking tool designed to test the performance of web servers by simulating client connections and request loads. It is lightweight, simple to use, and supports both static and dynamic page requests. It is widely used to evaluate web server performance and identify potential bottlenecks or areas for improvement.

It operates on a fork-and-exec model, which means that it creates multiple child processes, each simulating a client connection to the web server. This approach allows it to generate a large number of simultaneous connections and requests, effectively emulating the stress that a web server would face under real-world conditions.

### 5.1.2 Steps of Evaluation

To evaluate the performance of the high-performance web server using Web Bench, follow these steps:

- **Test Scenarios:** Design test scenarios to simulate various traffic loads and application use cases. Define the target URLs, request types (GET by default), and the number of clients for each scenario.
- **Test Execution:** Run Web Bench with the command-line arguments for each test scenario, specifying the number of clients and test duration. Execute the tests for both Reactor and Proactor based prototypes.
- **Data Collection:** Collect the performance metrics provided by Web Bench, such as queries per second, bytes transferred, and connection success rates, for each test scenario.
- **Data Analysis:** Analyze the collected data to compare the performance of the Reactor and Proactor pattern-based web servers under different load conditions.

## 5.2 Evaluation Metrics

When evaluating the performance of proposed web server, the following metrics should be considered:

- **Throughput:** A metric for throughput of the web server is QPS, it stands for queries per second. The QPS is particularly useful for assessing the capacity of a system, as it helps quantify how well the system is able to handle traffic and respond to requests. A higher QPS indicates that the system can handle more queries per second.
- **Error Rate:** Monitor the number of failed or erroneous requests as a percentage of the total requests. Lower error rates indicate better reliability and stability.

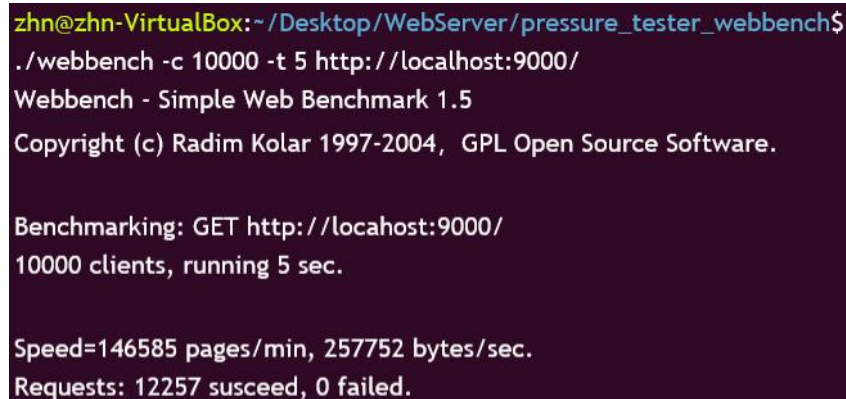


## 6 Evaluation Execution and Results

In evaluation execution, stress tests with 10000 simulated clients in 5 seconds for both prototypes were executed on a Ubuntu virtual machine.

Building and testing instruction please see in [Appendix 3](#).

### 6.1 Baseline Prototype Result

A terminal window with a dark purple background and light green text. The prompt is 'zhn@zhn-VirtualBox:~/Desktop/WebServer/pressure\_tester\_webbench\$'. The command entered is './webbench -c 10000 -t 5 http://localhost:9000/'. The output shows 'Webbench - Simple Web Benchmark 1.5', 'Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.', 'Benchmarking: GET http://localhost:9000/', '10000 clients, running 5 sec.', 'Speed=146585 pages/min, 257752 bytes/sec.', and 'Requests: 12257 succeed, 0 failed.'

```
zhn@zhn-VirtualBox:~/Desktop/WebServer/pressure_tester_webbench$
./webbench -c 10000 -t 5 http://localhost:9000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://localhost:9000/
10000 clients, running 5 sec.

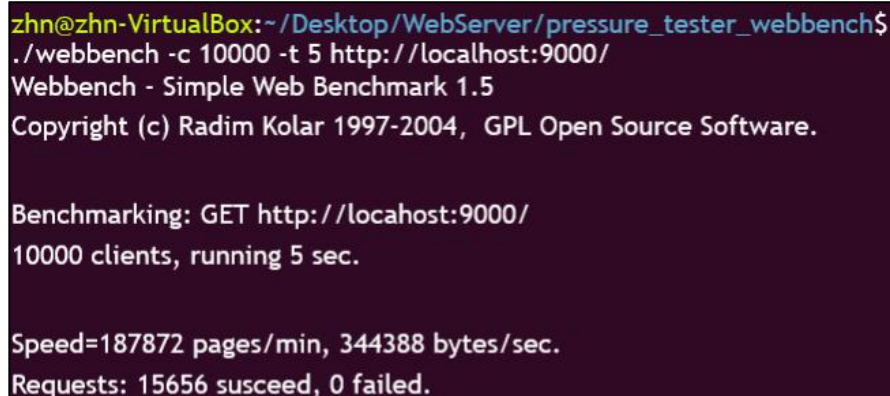
Speed=146585 pages/min, 257752 bytes/sec.
Requests: 12257 succeed, 0 failed.
```

Figure 5. Baseline Prototype Result

Total QPS for baseline prototype:

$$\text{QPS} = 12257/5 = 2451$$

### 6.2 Refined Prototype Result

A terminal window with a dark purple background and light green text. The prompt is 'zhn@zhn-VirtualBox:~/Desktop/WebServer/pressure\_tester\_webbench\$'. The command entered is './webbench -c 10000 -t 5 http://localhost:9000/'. The output shows 'Webbench - Simple Web Benchmark 1.5', 'Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.', 'Benchmarking: GET http://localhost:9000/', '10000 clients, running 5 sec.', 'Speed=187872 pages/min, 344388 bytes/sec.', and 'Requests: 15656 succeed, 0 failed.'

```
zhn@zhn-VirtualBox:~/Desktop/WebServer/pressure_tester_webbench$
./webbench -c 10000 -t 5 http://localhost:9000/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://localhost:9000/
10000 clients, running 5 sec.

Speed=187872 pages/min, 344388 bytes/sec.
Requests: 15656 succeed, 0 failed.
```

Figure 6. Refined Prototype Result

Total QPS for refined prototype:

$$\text{QPS} = 15656/5 = 3131$$

### 6.3 Analysis

Overall, refined prototype had a significant performance improvement by 27% than baseline prototype. This can be attributed to the better use of

asynchronous I/O operations and Linux kernel resource in Proactor pattern as previously discussed.

Asynchronous I/O operations enable the application to process additional requests while awaiting I/O completion. This enhances overall efficiency and helps prevent blocking, which may happen in the Reactor pattern. Besides, in Linux, the system kernel naturally excels at handling I/O-intensive tasks in comparison to threads, leading to a more effective use of the server hardware's CPU resources.

However, it is also essential to consider the specific use case, the server's requirements, and the environment in which the server will be deployed when making a decision. The Proactor pattern might have performed better in the stress test, but the reactor pattern could still be more suitable in situations with fewer concurrent requests or when simplicity and ease of implementation are prioritized.

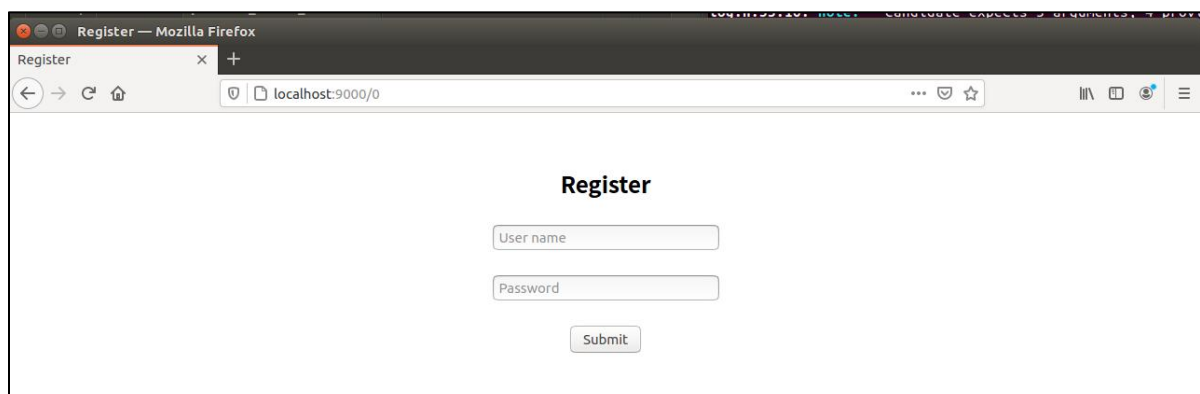
## 6.4 API Design

The HTML contents and the functionality of CGI scripts on proposed server are presented in this section.

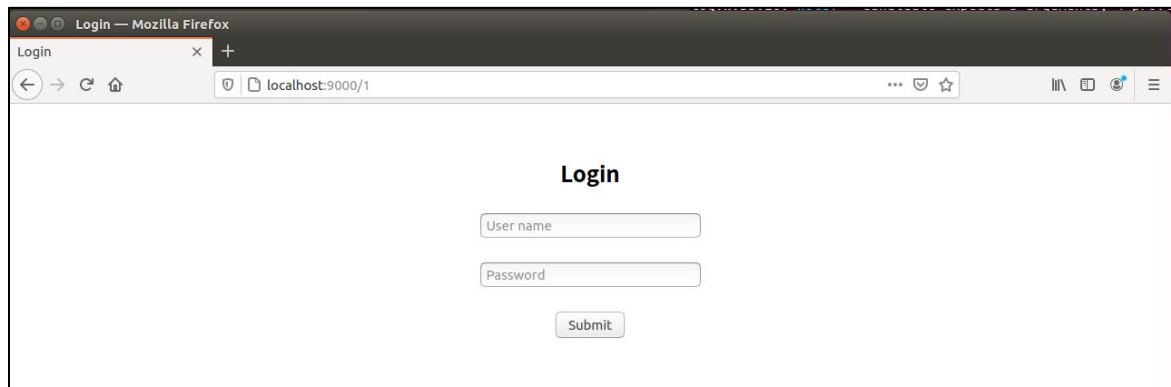
### 6.4.1 Home Page



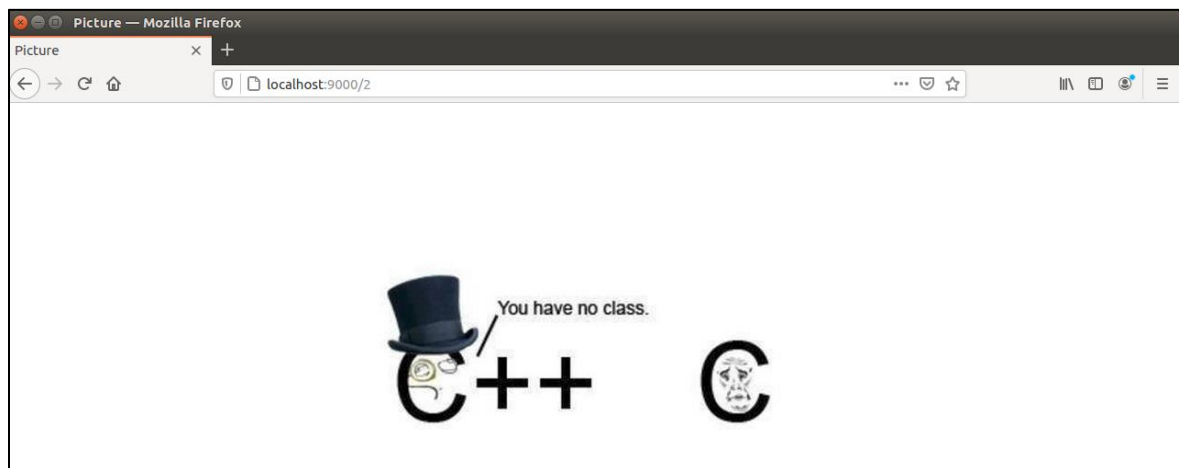
### 6.4.2 Register



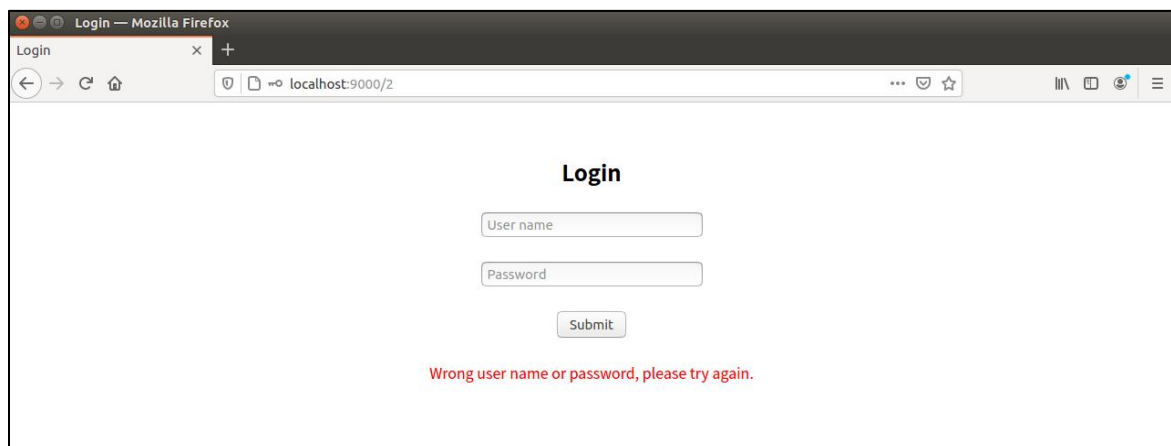
### 6.4.3 Login



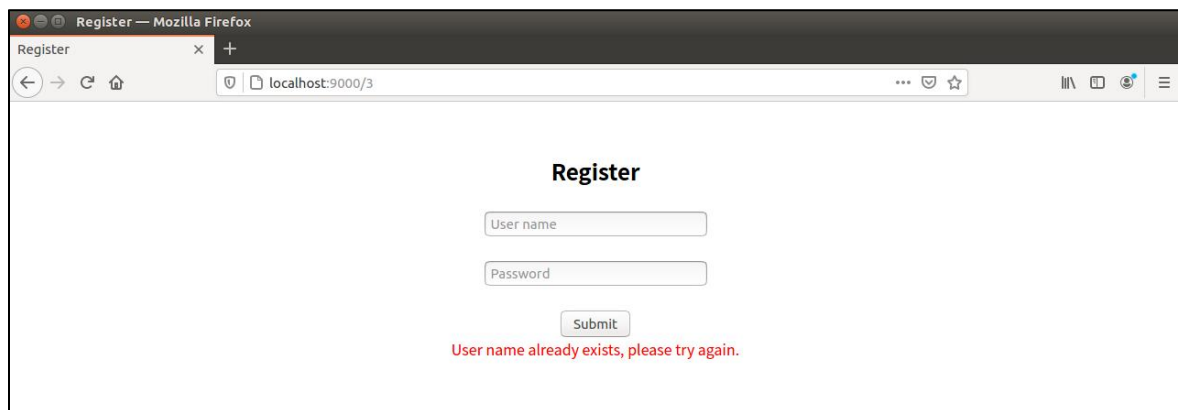
A picture presents after logging in.



When a wrong user name or password is entered, a warning appears.



When a duplicated username is entered in registering, a warning appears.



## 7 Concluding Remarks

One of the best parts of my project was the hands-on experience gained while implementing both the Reactor and Proactor patterns. Through these implementations, I learned a lot about the importance of non-blocking I/O operations in providing high-performance and scalable server applications. I realized that the reactor pattern is a simpler and more intuitive approach for handling multiple client connections but tends to suffer from performance issues as the number of connections grows. The proactor pattern, on the other hand, offers better scalability and resource management, but requires more complexity in implementation.

Several server programming related concepts were leveraged in my project, including:

- Event-driven Architecture: Both the reactor and proactor patterns are based on an event-driven architecture, where events (such as incoming client connections and requests) are processed as they occur.
- I/O Multiplexing: I used Linux's epoll API to perform I/O multiplexing, enabling efficient handling of multiple client connections simultaneously without blocking.
- Pooling: I utilized a thread pool to handle multiple requests concurrently, which helps in improving the server's throughput and reducing the response time.
- Internet Protocol: My project enhanced my understanding of two internet protocols: HTTP and TCP.

Finally, there are still ways in which the proposed web server can be improved to support new features and new capabilities. For example, the current thread pool implementation can be fine-tuned to dynamically adjust the number of threads, based on server load and resource availability. This will ensure optimal resource utilization and prevent performance degradation due to

contention or excessive context switching. Besides, implementing server-side caching mechanisms for frequently accessed data is also a good idea. It can reduce the need for redundant processing and improve response times.

## Appendix

### 1 Reactor Pattern (Copied from Assignment M4)

In reactor mode, the web server uses a event demultiplexer thread (typically main thread) to handle all incoming events, such as network connections and data transfers. This thread is responsible for registering new event to a eventnt dispatcher (usually an event table), and monitoring all events. When an event occurs, the main thread notifies the relevant server application to handle the event by calling a event handler (a worker thread) which was registered on the dispatcher, and inform this thread to read from socket and perform processing for the event, such as reading or writing data to the socket, and returns control to the thread to wait for the next event.

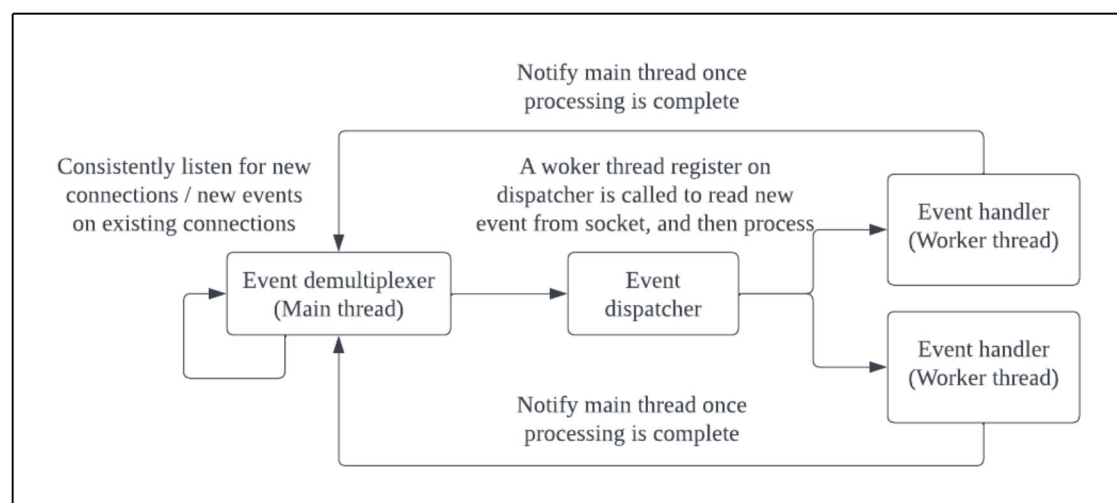


Figure . Architecture of the reactor event handling mode

The advantages of reactor mode include its simplicity and low overhead, as only the main thread is needed to read and dispatch all events to working thread. It is particularly useful when the number of connections is greater than the number of available threads or resources on the server. In this case, the reactor mode allows the server to handle the incoming connections more efficiently, as it can process multiple connections simultaneously using a single thread, without context switching or blocking.

### 2 Proactor Pattern (Copied from Assignment M4)

Proactor mode is another event handling pattern used in web servers to handle concurrent connections. The Reactor design pattern involves the event demultiplexer waiting for events to indicate when a file descriptor or socket is

ready for a read or write operation. The appropriate handler then performs the read or write operation.

In contrast, the Proactor pattern involves the handler, or the event demultiplexer acting on behalf of the handler, initiating asynchronous read and write operations. The actual I/O operation is performed by the operating system, which receives parameters such as the addresses of user-defined data buffers from which it retrieves data to write or to which it stores read data. The event demultiplexer then waits for events indicating the completion of the I/O operation and forwards them to the appropriate handlers.

The main advantage of proactor mode over reactor mode is the asynchronous I/O. In the Proactor pattern, I/O operations are performed asynchronously, the application can continue processing other requests while waiting for I/O to complete. This improves overall performance and can help avoid blocking, which can occur in the Reactor pattern.

Another advantage of proactor mode is its better utilization of OS and CPU resource, as it leave the processing of I/O operation to the OS. In Linux, the system kernel has various natural advantages in processing I/O intensive tasks compared with application threads. Therefore the CPU resource of the server hardware can be more efficiently utilized.

However, the proactor mode is more complex than reactor mode and requires more sophisticated programming techniques, such as thread pools, asynchronous callbacks, and completion handlers. This makes it more challenging to implement and maintain.

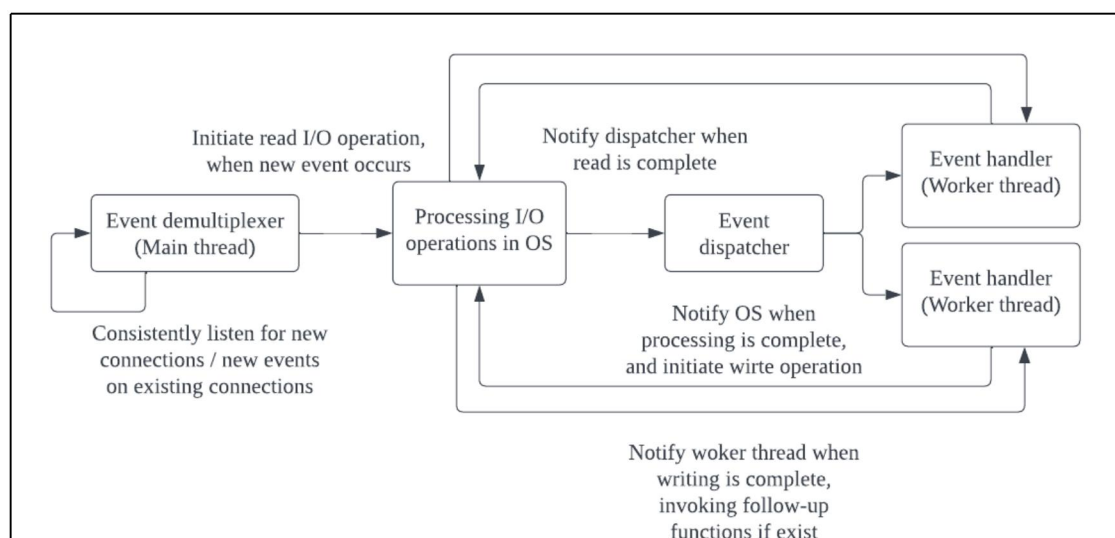


Figure . Architecture of the proactor event handling mode

Proactor leaves all I/O operations like reading/writing from/to the server's buffer, to main thread and the OS, worker threads only perform processing incoming events. The main idea of asynchronous I/O is to only notify other

components when the I/O or processing operation is finished. Therefore, the event demultiplexer will not keep states of each event and wait for it. It will move on to other events once initiating I/O operation and registering it to the OS is complete. Besides, the OS will be notified the location of the buffer where the I/O should start with.

After receiving a completion signal from OS with the information read from buffer, the dispatcher will call a corresponding worker thread to perform processing. Finally, the completion signal from worker thread will arrive at the OS for any follow-up procedure if exists, like closing the socket.

### 3 Building Instruction

#### 1. Develop & test environment

Ubuntu 20.04

MySQL 5.7.30

FireFox browser

Webbench 1.5

#### 2. Create and use MySQL database

```
mysql -u root -p
```

```
mysql -> CREATE DATABASE mydb;
```

```
USE mydb;
```

```
CREATE TABLE user( username char(50) NULL, passwd char(50) NULL )
```

```
ENGINE=InnoDB;
```

```
INSERT INTO user(username, passwd) VALUES('name', 'passwd');
```

#### 3. Modify line 123 in main.cpp

```
From connPool->init("localhost", "root", "Aa199781.", "mydb", 6000, 8);
```

```
To connPool->init("localhost", "root", "<your root server password>", "mydb",  
6000, 8);
```

#### 4. Modify line 21 in http\_handler.cpp

```
From const char* doc_root = "/home/zhn/Desktop/WebServer/resource";
```

```
To const char* doc_root = "<your directory of resource file>";
```

#### 5. Make file and run server in terminal

```
Make server
```

```
./server port
```

#### 6. Input URL on browser

localhost:port

## 7. Test with webbench1.5

```
gcc webbench.c -o webbench
```

```
./webbench -c 5000 -t 5 http://localhost:9000/
```

// Where -c is number of clients, -t is time in seconds

## Reference

1. Soni, R., & Soni, R. (2016). Nginx Core Architecture. Nginx: From Beginner to Pro, 97-106.
2. Stevens, W. R., & Narten, T. (1990). UNIX network programming. ACM SIGCOMM Computer Communication Review, 20(2), 8-9.
3. Kerrisk, M. (2012) The linux programming interface: A linux and UNIX system programming handbook. San Francisco, CA: No Starch Press.
4. Kunda, D., Chihana, S., & Sinyinda, M. Web Server Performance of Apache and Nginx: A Systematic.
5. Wehrle, K., Pahlke, F., Ritter, H., Muller, D., & Bechler, M. (2004). Linux Network Architecture.
6. (Chinese book) You, S. (2013). Linux Gao Xing Neng Fu Wu Qi Bian Cheng = high performance linux server programming. Beijing: Ji xie gong ye chu ban she.
7. Comer, D. E., & Stevens, D. L. (1993). Internetworking with TCP/IP Vol. III: Client-server programming and applications. Prentice-Hall, Inc..
8. My Assignment M4: A Web Server Architecture for High Concurrency