Study and Optimization of Server Load Capacity in High Concurrency Scenarios

Hui Wang
School of Computer Science and Engineering
Xi’an Technological University
Xi’an, China
E-mail: wanghuihy@xatu.edu.cn

Le Qiang
School of Computer Science and Engineering
Xi’an Technological University
Xi’an, China
E-mail: qiangle@st.xatu.edu.cn

Jiasheng Wei
School of Computer Science and Engineering
Xi’an Technological University
Xi’an, China
E-mail: weijiacheng@st.xatu.edu.cn

Haoyu Li
Faculty of Transportation Engineering
Kunming University of Science and Technology
Kunming China
E-mail: haoyumli@gmail.com (H.Y.)

Teng Yan
School of Computer Science and Engineering
Xi’an Technological University
Xi’an, China
E-mail: yanteng@st.xatu.edu.cn

***Abstract*—In high-concurrency scenarios, network and disk I/O-intensive operations often compete for shared resources, resulting in a decline in the server's load capacity. To address this challenge, this paper proposes a sophisticated high-concurrency server optimization solution. It utilizes various Reactor models in the Linux system, combined with the powerful Epoll mechanism and thread pool, to conduct research and optimization on the server's load capacity.Firstly, the event-driven and other modules required by the Web server are implemented and integrated. Secondly, the number of Reactors, the number of threads, and the business processing time under the Linux system are designed and controlled, and the design and implementation scheme of the high-concurrency server based on the Reactor model with the Epoll mechanism and thread pool are determined. Finally, the performance differences and the best usage scenarios of Web servers with different Reactor models in high-concurrency environments are analyzed through stress tests. The comparison results show that the QPS (Queries Per Second) indicator of the Web server based on the multi-Reactor multi-thread model is three times higher than that of the single-Reactor single-thread Web server, verifying its overall advantages in high-concurrency and long-term business processing. The research results demonstrate the applicable scenarios of different Reactor models, providing theoretical basis, implementation examples, and data support for choosing the appropriate Reactor model in actual server development, helping developers select the most suitable Reactor model according to specific server requirements to ensure higher efficiency in high-concurrency scenarios.**

Keywords-component; High Concurrency；Load Capacity；Thread Pool；Reactor model；Web server

I. INTRODUCTION

In the context of the Internet era, which is characterized by an exponential growth in the number of websites and server applications, there has been a significant increase in the volume of data accessed via these platforms. This considerable rise in demand for server resources has consequently resulted in a notable escalation of the load pressure experienced by these servers.

In instances where multiple users access the server simultaneously, the resulting burden on the server's resources is likely to lead to a decline in performance, as well as potential server downtime. An efficacious solution to this problem is the construction of web servers that are capable of effectively managing high concurrency scenarios. The fundamental technology underlying web servers, specifically network programming, has a direct impact on the overall performance of the system.

The most commonly utilized web programming models are the Reactor and Proactor models [2], with the Reactor model being further subdivided into three types based on the number of processes or threads employed. It is important to note that different network programming models are suitable for web servers under varying operational scenarios. Therefore, the aim of this thesis is to investigate the load capacity of servers utilizing different Reactor models in high concurrency scenarios, while also analyzing and summarizing the performance differences between these models. Additionally, the thesis seeks to provide theoretical foundations, implementation examples, and data support for selecting appropriate Reactor models for actual server development. The main work of this article includes:

The objective is to implement the web server modules within the Linux system, which encompasses the following components: event driver, network connection, network processing, thread pool, socket, event, and asynchronous driver.

Each module will be integrated, with the Epoll mechanism and thread pool identified as the foundation for the design and implementation of a high concurrency server based on the Reactor model. Subsequently, the web server will be constructed according to the Reactor model through various combinations and parameter adjustments.

The Web bench stress test tool [3] will be employed to conduct stress tests on web servers utilizing different Reactor models within a highly concurrent environment. The performance of the three Reactor models will then be analyzed using MATLAB, thereby providing a reference for the selection of an appropriate web server programming model based on the performance outcomes of these Reactor models in high concurrency situations.

The remaining sections of this paper are organized as follows:

Section II introduces the relevant technical foundations. Section III introduces the specific design and implementation ideas of the scheme. Section IV introduces the experimental design and result analysis. The conclusions and future work are discussed in Section V.

II. GUIDELINES FOR MANUSCRIPT PREPARATION

A. Technical Architecture for High Concurrency Scenarios

The concurrent access of a large number of users places significant pressure on the server's data exchange and processing capabilities. In order to ensure the successful completion of business operations, a variety of highly concurrent processing techniques have emerged, which are tailored to different application scenarios and present a range of technical architectures. In the context of the civil aviation passenger service information system, Li Yongjin et al. proposed a phased event-driven architecture with the objective of enhancing the system's capacity to process highly concurrent requests [4]. In a further development of WeChat, Li et al. employed Redis caching technology to enhance the system's concurrency [5]. Yuntao Xu et al. employed the use of a Nginx reverse proxy, along with techniques such as multi-processing, multi-threading, and multi-core, with the objective of accelerating the parallel search process in a highly concurrent iris recognition system [6]. Li Junfeng et al. conducted a comprehensive and effective analysis of the high concurrency issue in the airline ticket reservation system, identifying potential solutions through load balancing, page optimization, cache design optimization and database optimization [7]. Wang Jiye et al. achieved high concurrency in the processing of large-scale heterogeneous sensory data, including reception, parsing and distribution, through the utilization of bespoke data structures and asynchronous I/O multiplexing [8]. In addition to such application-specific high concurrency solutions, numerous scholars have also conducted research and advocated for mainstream high concurrency processing techniques. For example, Yannan Wang et al. delineated high concurrency optimization solutions from five perspectives: the web application front-end, back-end program code, database, web application middleware configuration and server load [9]. Kewei Li provided an overview of high concurrency processing techniques for network links, reverse proxies, application services, data caching, and databases, respectively, in Internet distributed architectures [10].

B. multi-threading technology

In addition to enhancing the concurrency of the system through the utilization of hardware technology, prominent web servers such as IIS, Apache, and Tomcat respond to a considerable number of concurrent requests through the implementation of a multi-threading mechanism [11, 12].

In the literature, Bojin Sun et al. put forth a solution to the resource occupancy and contention problems through the implementation of algorithms, data structures, optimized interrupts, and optimized process and routine scheduling. The literature [14] describes the factors that play an important role in the performance of web servers and proposes a new thread-based server architecture. Asynchronous programming enables the development of services capable of handling millions of requests without saturating memory and CPU utilization, thereby enhancing the I/O capabilities of server systems. Karsten M. et al. present the design and implementation of a flexible user-level M: N threading runtime, constructed from scratch, which has been developed to achieve these objectives in [16]. The system's principal components are efficient load balancing and user-level I/O blocking. To address the issue of threads being affected by blocking anomalies, namely the loss of parallelism when executing blocking system calls, which leads to low kernel utilization and unnecessarily high response times, Florian Schmaus et al. introduced pseudo-blocking system calls based on modern asynchronous queuing system call techniques (e.g., Linux's io_uring) in the literature [17] in order to circumvent these anomalies. Techniques such as Nginx and Keepalived are frequently employed to address the load challenges encountered by highly concurrent applications. Literature [18] assesses the performance of a server cluster environment based on Nginx and Keepalived, evaluates the efficacy of Nginx-based algorithms such as WRR, IP_HASH and LEAST_CONN, and designs and optimizes the IP_HASH algorithm.

C. Reactor model

The Reactor model represents an event-driven design pattern that is widely employed in web server development with the objective of creating highly concurrent and high performance web applications. The fundamental concept is to monitor and disseminate input/output (I/O) events via an event distributor (Reactor) and relay the events to the designated event processor for processing. The Reactor model attains efficient concurrent processing through non-blocking I/O and event notification mechanisms, rendering it suitable for highly concurrent, low-latency application scenarios.

The Reactor model may encounter performance bottlenecks and scalability challenges when processing a large number of concurrent connections. The performance and scalability of the Reactor model may be enhanced through the optimization of event processing, scheduling algorithms and I/O multiplexing techniques. The advancement of asynchronous programming and co processing technology has led to the introduction of asynchronous I/O, co-processing scheduler, and other related technologies [21], which have further enhanced the concurrent processing capability and resource utilization of the Reactor model. The Reactor model is currently employed not only in the development of standalone web applications, but also in the context of distributed systems. By designing a distributed event-driven architecture and optimizing the message passing mechanism, the Reactor model is employed in the construction of a highly reliable and high-performance distributed system [22].

The literature [19] examines the advantages, limitations, and applicable scenarios of the Reactor model through an analysis of the event-driven programming model, the design of the event loop, and the registration and distribution of event processors. With regard to the Reactor programming model, domestic and foreign research institutions and enterprises have developed numerous frame-works and libraries with the objective of simplifying the development and maintenance of event drivers. For example, Node.js and Netty [20] represent such frameworks.

The Reactor model typically comprises an event distributor, which monitors input/output (I/O) events and distributes them to the relevant event handlers upon occurrence, and an event handler, which handles the components of a specific I/O event. Event handlers are registered with the event distributor and are invoked upon the occurrence of an event.Three principal models for Reactor modelling in web servers have been identified: the single Reactor single-threading model, the single Reactor multi-threading model, and the multi-Reactor multi-threading model. The suit-ability of different Reactor models for use in various web server environments is a key consideration.

The single Reactor single-threading model represents the most fundamental iteration of the Reactor model, characterized by a straightforward structure, making it well-suited to scenarios involving a limited number of I/O events. The single Reactor multi-threading model builds upon the single Reactor single-threading model by incorporating a thread pool to facilitate event handling and enhance concurrent processing capabilities. The multi-Reactor multi-threading model represents a further optimization of the Reactor model, whereby the concurrent processing capability is enhanced by the distribution of the task of event listening and distribution among multiple Reactor instances. The system is typically constituted of a master Reactor, which listens for connection requests and distributes new connections to slave Reactors. Each slave Reactor runs in a separate thread and listens for I/O events on its respective connection. The slave Reactor then disseminates the events to the worker threads within its management thread pool for processing.

The implementation of multi-Reactor multi-threading represents a departure from the single Reactor multi-threading approach. In this new approach, the Reactor component has been decoupled from a single Reactor module, and instead, it is comprised of a master Reactor module and multiple slave Reactor modules. Concurrently, the original discrete Reactor module is tasked with event listening and distribution, but has also been divided into a master Reactor and a slave Reactor module [24]. The master Reactor module listens for events and disseminates them to a slave Reactor, which oversees the events assigned to it and handles them with an event handler. Multi-Reactor multi-threading is illustrated in Figure 1.

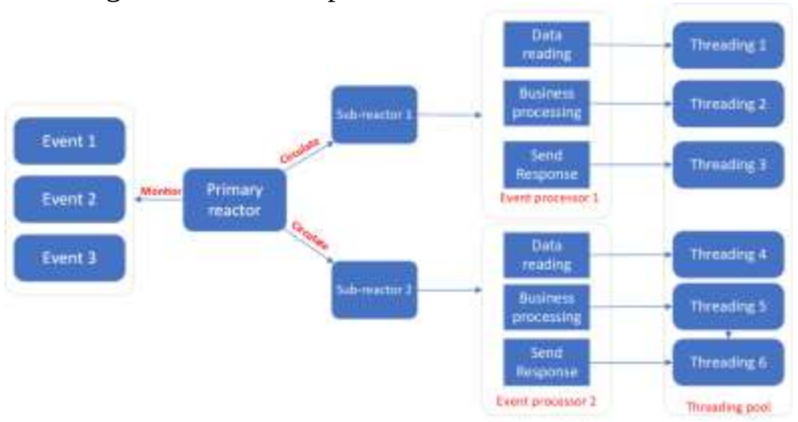


Figure 1. Reactor model based web server

III. Design Ideas And System Construction

In the Linux system, a web server is designed to control the number of Reactor threads and business processing time, among other variables, in order to simulate different Reactor models. The single Reactor single-threading server serves as the foundation for the subsequent stages of development, beginning with its construction and subsequent evaluation in a highly concurrent environment. This process entails the identification of shortcomings and the implementation of improvements and optimizations to enhance the server model to a single Reactor multi-threaded. The subsequent phase involves rigorous testing and the eventual realization of the multi-Reactor multi-threading model. Finally, the data obtained from the testing phase is subjected to comprehensive analysis and synthesis. Subsequently, the data obtained from the testing of all models will be analyzed and summarized, after which the load capacity of the server in high concurrency scenarios will be studied.

The specific implementation ideas of this paper's scheme are as follows: firstly, the various modules of the web server must be implemented, including the event driver module, network connection module, network processing module, thread pool module, socket module, event module and asynchronous driver module. Secondly, the thesis employs a thread pool to manage multiple threads, thereby enabling the server to transition between a single-threading and a multi-threading model by activating or deactivating the thread pool. Subsequently, the maximum number of concurrently active threads that the thread pool can accommodate is determined by setting the number of threads, thereby enabling an investigation into the impact of varying thread numbers on the performance of a multi-threading web server under high concurrency. Ultimately, the modules are integrated to create three reactor models of web servers through a combination of disparate configurations and parameter modifications.

A. *system architecture*

Event-driven programming is employed in the construction of high-performance Web servers. These servers are designed to remain continuously attentive to network connection requests, and upon the establishment of a connection, the server will initiate the relevant event processing function to facilitate the processing of the network connection. Once the process of establishing a network connection has been completed, the client socket must be obtained and the corresponding event handler must be triggered in order to read the information from the client. Subsequently, the event handler is triggered in order to respond to the client request.

The Reactor model represents the implementation of event-driven programming concepts within a Web server design pattern. The system is equipped with an event loop, which is responsible for listening to and distributing events. Upon the occurrence of an event, the relevant processor is duly informed and tasked with handling the event.

The Reactor module of the server is implemented by the event driver as the core component. The Reactor module is designed to constantly listen for client requests. Upon detecting a client request, the Reactor is called to handle the connection request event. This enables the establishment of a connection between the server and the client through the Linux kernel, as facilitated by the API for the configuration of client sockets. Consequently, the Reactor module represents the connection between the two entities. Subsequently, the Reactor module will process further events pertaining to the connected client, calling upon the module that has been specifically designed to serve the client in question with the requisite services. Subsequently, the Reactor module will persist in monitoring for new client connection requests and disseminating them to clients who have successfully established a connection. To facilitate the module's service to the client, it will receive the data from the client through the API provided by the Linux kernel, generate the data corresponding to the client after service processing, and transmit the generated data to the client through the API provided by the Linux kernel once more. Figure 2 illustrates the configuration of the Reactor model web server.

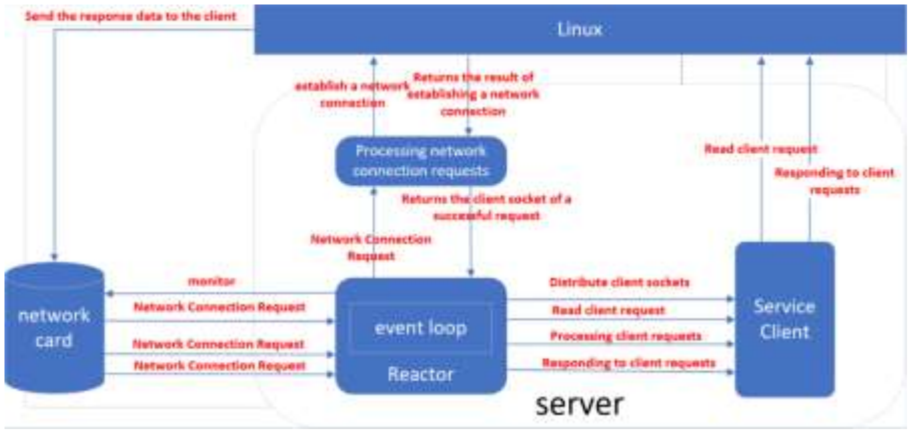

Send the response data to the client
Linux
establish a network connection
Returns the result of establishing a network connection
Processing network connection requests
Read client request
Responding to client requests
Returns the client socket of a successful request
Network Connection Request
monitor
Network Connection Request
Network Connection Request
Network Connection Request
network card
event loop
Reactor
Distribute client sockets
Read client request
Processing client requests
Responding to client requests
Service Client
server

Figure 2. Reactor model based web server

Reactor implementations The development of the Web server is carried out under Linux, and the implementation of the server's Reactor uses Epoll, an I/O multiplexing interface provided by Linux.

Epoll is an I/O event notification mechanism provided by Linux to listen for the occurrence of events registered on Epoll. When Epoll is enabled, the programme blocks until an event occurs in the listening event, and then it returns to that event. This mechanism allows for efficient I/O multiplexing and is often used to build high-performance web servers.

The Reactor module uses Epoll as an event loop for registering, listening, and distributing events, as shown in Figure 3.

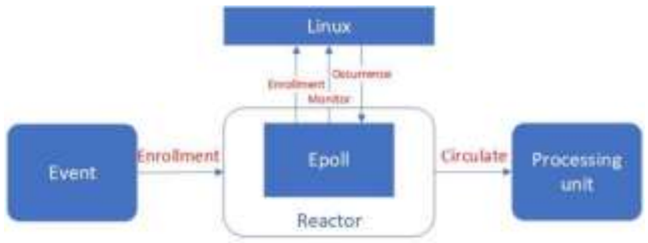


Figure 3. Reactor Modules

Given that Reactor is based on an event-driven model, it can be seen that the key to this model lies in the occurrence and processing of events. Furthermore, given that the web server handles multiple network connection requests simultaneously, it can be seen that this is also a key factor in the event-driven model. The server initially listens to the first client and establishes a connection (listen event), then performs a read event on the first client (read event 1), and finally executes a business process (business process 1) subsequent to the completion of the read event. At this juncture, the second client requests a connection from the server. Consequently, two events are registered on Epoll: the listen event and the write event 1. The order of execution of these two events is indeterminate. The server may execute the listen event first, after which the read event 2 is registered on Epoll. At this juncture, two events are registered on the Epoll (write event 1 and read event 2). The order of execution of these two events is also unknown. This is followed by the occurrence of all subsequent events.

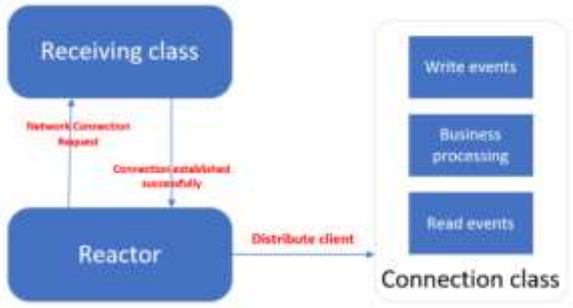


Figure 4. Reactor Modules

The actual test is conducted in a highly concurrent environment, wherein thousands of clients are accessing the web server simultaneously. Consequently, the order of events processed by the server is entirely random. With regard to the type of processor employed in the project, please refer to Figure 4.

B. system implementation

In the implementation of a web server based on disparate Reactor models, it is first necessary to construct the server modules that would be typical of a generic web server. In addition, the specific modules required for each Reactor model must be developed. These include the Reactor module, which implements the Reactor model, and the thread module, which implements the Thread model. In essence, the server implements the web server for different Reactor models by enabling the requisite modules.

With regard to the server module, the fundamental component of the server as a Reactor model is Reactor. Given that Reactor is based on event-driven concepts, it is essential to implement the event-driven module as a preliminary step, and subsequently construct the server module on the foundation of the event-driven module.

The Event-Driven module employs the Event-Driven class as the fundamental component of the event driver, the Epoll class as the implementation of the Event-Driven class for the event driver, and the Event class as the abstract conduit for event interaction with the Event-Driven module to facilitate registration, listening, and updating of events.

1) Event Driver Module

The event-driven module comprises the Event-Driven, Epoll and Event classes, as illustrated in Figure 5.

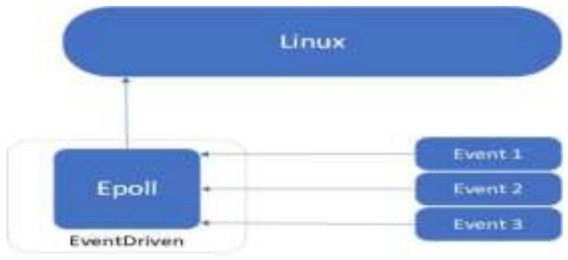


Figure 5. Event Driven Modules

The Epoll class represents an encapsulation of the epoll file descriptor. Epoll is responsible for the registration and updating of events, which it achieves by interacting with the Linux system. Additionally, Epoll listens for the triggering of events. The Event class provides an abstraction of events. The properties of events can be controlled, and interaction with the Epoll class can be facilitated through the Event-Driven class, which is used for the registration and updating of events. The Event-Driven class encapsulates the Epoll class and is employed to initiate the event loop, awaiting the occurrence of events that can be accessed.

a) Event-Driven Class

The Event-Driven class is responsible for event driving, whereby epollfd is employed to encapsulate the I/O multiplexing interface Epoll, provided by Linux. Update Event is used to call the epollfd function of the same name, with the objective of updating the event. The Event-Driven class is illustrated in Figure 6.

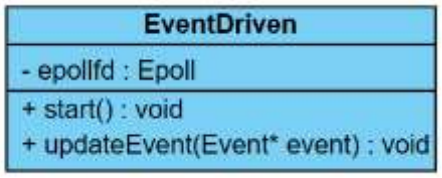


Figure 6. Event-Driven Class

b) Epoll Class

The Epoll class encapsulates epoll, which interacts with the Linux system for the purpose of event management. The updateEvent function is employed for the purpose of registering or updating events, while the wait function is used for the purpose of waiting for a registered event to be triggered. The Epoll class is illustrated in Figure 7.

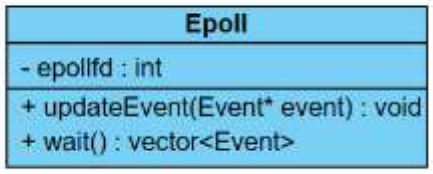


Figure 7. Epoll Class

c) Event Class

The Event class provides an abstraction of an event, utilizing the fd file descriptor to represent a specific event object. It also employs the use of event, inEpoll and callback to represent the type of the current event, the registration status of the event and the event handler function to be executed for the event, respectively. The ed variable is used to associate the Event with Event-Driven, while the latter is used to associate an Event with an Event-Driven. Additionally, ed is employed to pass the Event itself as a parameter to the Event-Driven, thus enabling the execution of the updateEvent function.

The function comprises several parts, including those that set the current event as a listen, read or write event, close the event, check or set whether it is registered, obtain a file descriptor, execute the event handler and set the event handler. The implementation of the code constitutes a direct call to the relevant API provided by the Linux system. Consequently, the logic related to the execution of the program is not presented, as it is not pertinent to this discussion. The Event class is illustrated in Figure 8.

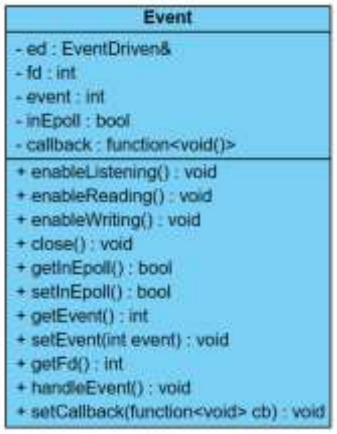


Figure 8. Event Class

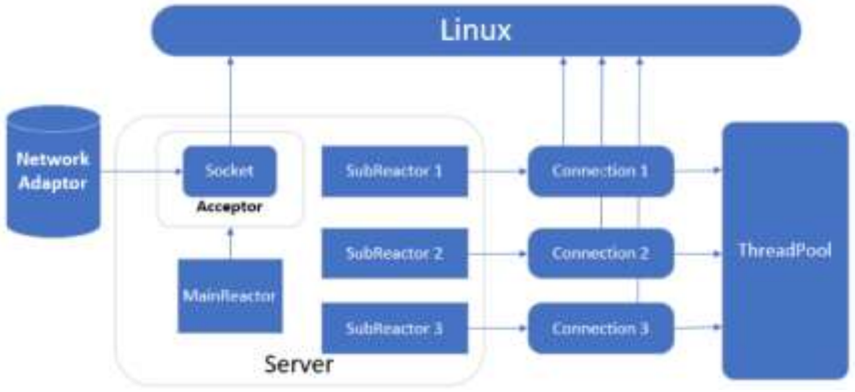


Figure 9. Server Module

2) Server module

The server module serves as the foundation for the implementation of all Reactor-type servers in the thesis. By adding or removing specific modules, it is possible to create different Reactor model web servers. Figure 9 illustrates the server module.

He fundamental component of a server based on the Reactor model is the Reactor module, which is founded upon the principle of event-driven processing. Consequently, the thesis employs the Event-Driven class as the actual Reactor module. The functionality of registering, listening and distributing events inherent to Reactor is achieved through the interaction of the Event-Driven class.

With the Epoll class and the Event class. In the case of the multi-Reactor model, the use of multiple Reactor modules is necessary, with these being divided into Master Reactor and Slave Reactor for different purposes.

The server's function for receiving client requests and establishing connections is designed as the Acceptor class, which interacts with the Linux kernel through the socket class that abstracts sockets to establish a connection with the client.

The Connection class is used to provide specific services to the client, including reading the client's request message, processing the business logic and generating a response message and returning it to the client. In the case of servers operating in a multi-threading mode, the thread pool module is employed.

a) Server Class

The Server class provides the external framework for the entire server model, which serves as a user-facing object and offers methods for initializing the server and initiating the various Reactor models, as illustrated in Figure 10.

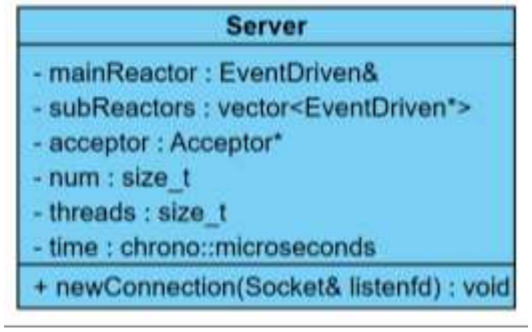

Server
- mainReactor : EventDriven&
- subReactors : vector<EventDriven*>
- acceptor : Acceptor*
- num : size_t
- threads : size_t
- time : chrono::microseconds
+ newConnection(Socket& listenfd) : void

Figure 10. Server Class

The main Reactor module, designated as "main-Reactor," is the primary component of the server. It is responsible for registering, listening to, and distributing all events within the single Reactor model. In contrast, the sub-Reactors module, or "sub-Reactors," operates as a slave Reactor module of the server. In the single Reactor model, it is effectively null. In the multi-Reactor model, the reactor model is empty and receives and processes clients distributed by the main-Reactor. The acceptor is used as an interface to access the Acceptor class. The num parameter is used to specify the number of sub-Reactors, while the thread parameter is used to specify the size of the thread pool. The time parameter is used to specify the length of the business processing time. The new-Connection parameter is used to specify the number of sub-Reactors and the length of the business processing time. The length parameter is used to establish a new client connection.

The server is initiated through the constructor, which first establishes the parameters provided by the user, defining the number of reactors, the number of threads, and the business processing time. Secondly, the acceptor class is initialized, thereby establishing the manner in which the server will handle client connections. Ultimately, the number of reactors serves to distinguish between a single reactor model and a multi-reactor model for the server. In the event that the model is that of a single reactor, the process concludes directly, with the initialization of the slave reactor. The flowchart of the constructor is presented in Figure 11.

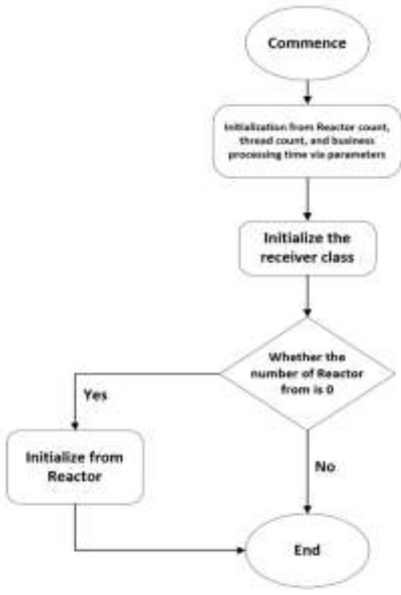

Commence
Initialization from Reactor count, thread count, and business processing time via parameters
Initialize the receiver class
Whether the number of Reactor from is 0
Yes
Initialize from Reactor
No
End

Figure 11. Flowchart of The New-Connection Function

The flowchart of the newConnection function is presented in Figure 12. The responsibility for establishing the connection is allocated to either the slave or the master reactor, depending on whether multiple reactors are employed.

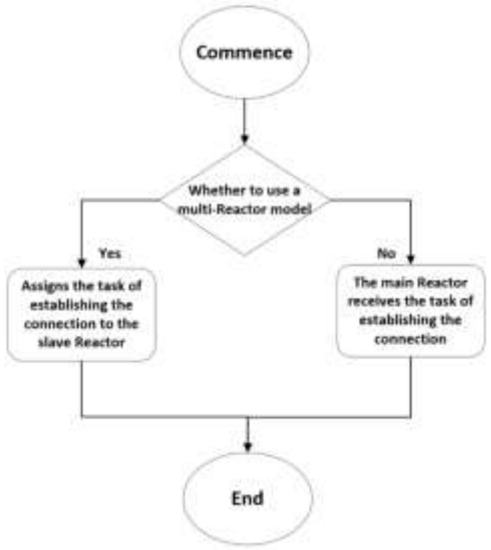

Commence
Whether to use a multi-Reactor model
Yes
No
Assigns the task of establishing the connection to the slave Reactor
The main Reactor receives the task of establishing the connection
End

Figure 12. Epoll Class

b) Socket class

The Socket class is analogous to the Event class in that it represents an abstraction of a particular entity, as illustrated in Figure 13. The Socket class provides an abstraction of sockets, where fd represents the file descriptor created by the Linux system to represent the socket. As with the Event class, the functions of the Socket class are all calls to the Linux system APIs, and there is no need to specify the implementation.

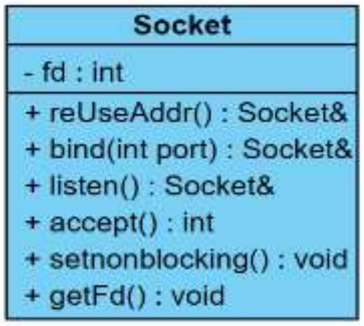


Figure 13. Socket Class

c) Acceptor Class

The Acceptor class is employed to receive new client connection requests and establish a connection, as illustrated in Figure 14. The listen fd socket is utilized as a listening socket to monitor client requests directed towards the server. The event variable represents the current event type, while the callback variable represents the event handler. The setCallback variable represents the event handler. The acceptConnection function is employed to create a connection, which constitutes the primary call listenfd accept function. This is achieved through the utilization of the Linux system API to establish a connection, without the provision of a specific code demonstration.

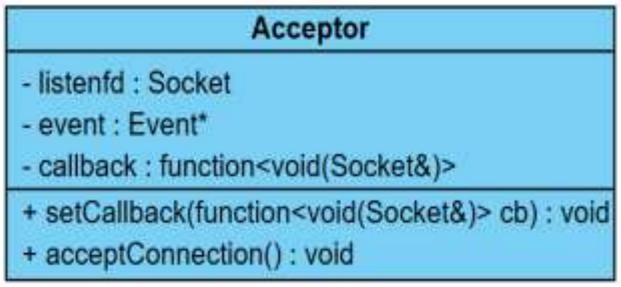

Acceptor
- listenfd : Socket
- event : Event*
- callback : function<void(Socket&)>
+ setCallback(function<void(Socket&)> cb) : void
+ acceptConnection() : void

Figure 14. Acceptor Class

d) Connection Class

The Connection class is employed for the purpose of serving clients, as illustrated in Figure 15.

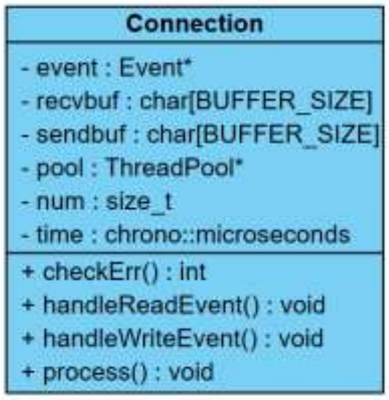

Connection
- event : Event*
- recvbuf : char[BUFFER_SIZE]
- sendbuf : char[BUFFER_SIZE]
- pool : ThreadPool*
- num : size_t
- time : chrono::microseconds
+ checkErr() : int
+ handleReadEvent() : void
+ handleWriteEvent() : void
+ process() : void

Figure 15. Connection Class

The event represents the current event type. The recvbuf is used to receive the client request message, while the sendbuf is used to send the response message to the client. The pool is the thread pool, while the num is the size of the thread pool. The time is the transaction processing time, and the checkErr is a utility function used to check whether some API calls of the Linux system have returned an error. Finally, the handleReadEvent is used to handle read events, while the handleWriteEvent is used to handle write events. The process is employed for business processing in order to implement the client request. The handleReadEvent is utilized for the handling of read events. The handleWriteEvent is employed for the handling of write events. The process is used for business processing with the objective of fulfilling the client's request. The constructor of Connection will set the event type to read, in addition to initializing the members and binding the event handler to the handleReadEvent. The handleReadEvent is employed for the execution of the read event. With regard to the implementation of the handleReadEvent and handleWriteEvent functions, a number of approaches may be adopted with regard to the reading or writing of data to the buffer. The focus of this paper is on the manner in which the server assigns transactions to the worker threads of the thread pool in a multi-threading model. The flowchart of the handleReadEvent function is shown in Figure 16.

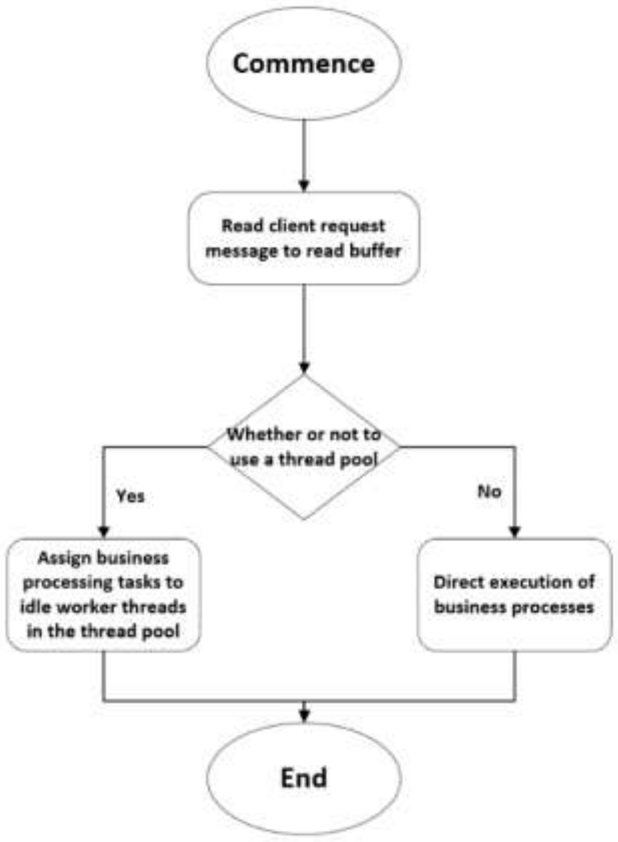

Commence
Read client request message to read buffer
Whether or not to use a thread pool
Yes
No
Assign business processing tasks to idle worker threads in the thread pool
Direct execution of business processes
End

Figure 16. Flowchart of the handle-ReadEvent

Firstly, the message sent by the client is read into a read buffer for parsing and subsequent specific business processing. Secondly, it determines whether the thread pool is to be employed. In the event that the thread pool is to be utilized, the business processing tasks are assigned to the thread pool for completion. Otherwise, the business processing tasks are continued to be executed in the current thread.

e) ThreadPool Class

Figure 17 illustrates the thread pool utilized by the ThreadPool class as a server. The term "works" encompasses all worker threads, while "tasks" represents the functions to be executed. The "mutex" is employed to guarantee synchronization between threads, and the "condition condition variable" is utilized to notify if a new function task has been queued. The "stop" indicator determines whether to halt the pool, and the "enqueue function" is utilized to receive a function to be added to tasks awaiting execution. A function is appended to the list of tasks that are to be executed.

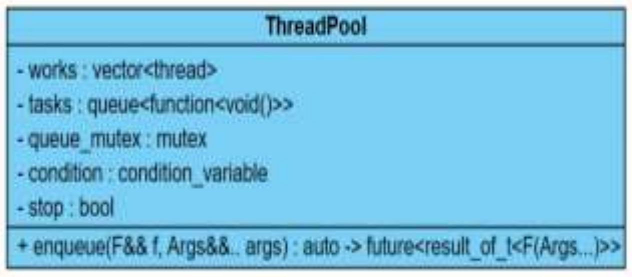

ThreadPool
- works : vector<thread>
- tasks : queue<function<void()>>
- queue_mutex : mutex
- condition : condition_variable
- stop : bool
+ enqueue(F&& f, Args&&... args) : auto -> future<result_of_t<F(Args...)>>

Figure 17. ThreadPool Class

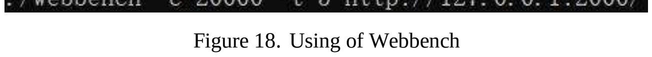
IV. SIMULATION EXPERIMENTS AND ANALYSES

A. Introduction to the experimental environment

The hardware configuration of the test equipment described in the thesis is as follows: the processor is an AMD Ryzen 7 4800H with 8 cores and 16 threads; the memory size is 16 GB at 3200 MHz; and the network card is an Intel® Wi-Fi 6 AX200 160 MHz. The operating system used for the test is Arch Linux.

The test environment is a high concurrency network test environment, utilizing the Webbench stress test tool for high concurrency testing. Webbench is capable of simulating multiple concurrent clients, sending HTTP requests to the server in order to evaluate the server's QPS performance. QPS, Queries Per Second, is a significant index of server performance. It represents the number of network requests that can be processed per second on a web server, and is specifically employed to gauge the server's capacity to withstand high concurrency.

The Webbench test of the web server utilizes a uniform client concurrency of 20,000, requesting the web server's static pages, with an HTTP request time of 5 seconds. The parameters are set in accordance with the specifications outlined in Figure 18.



```
./webbench -c 20000 -t 5 http://127.0.0.1:2000/
```

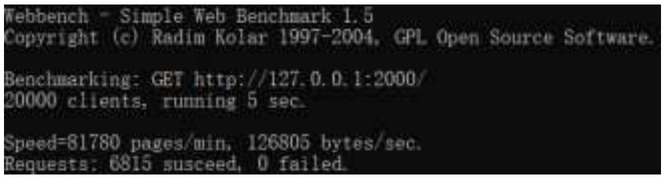
Figure 18. Using of Webbench

A detailed account of the startup options employed by Webbench is presented in Table 1.

TABLE I. EXPLANATION OF WEBBENCH USAGE OPTIONS

Webbench Usage Parameters	Parameter explanation
./webbench	Starts the Webbench testing tool.
-c	Specify the number of concurrent clients
-t	Specify the duration of HTTP requests
http://127.0.0.1:2000/	accesses the specified web server

The test results for the web server, as displayed in Figure 19, indicate a speed of 81,780 pages per minute and 126,805 bytes per second. The value following 'Speed' represents the number of bytes processed by the web server per second; however, this is not employed as an indicator in the thesis. The value following the designation "Speed" represents the number of bytes per second that are processed by the web server.



```
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://127.0.0.1:2000/
20000 clients, running 5 sec.

Speed=81780 pages/min, 126805 bytes/sec.
Requests: 6815 susceed, 0 failed.
```

Figure 19. Webbench Test Result Chart

A significant variable in the experimental design is the time required for business processing. The business processing time is employed in the modelling of the type and size of network requests. To illustrate, if a client requests a static web page from the server and requests the server to download a file from the server, the input/output (I/O) time spent is different. In most cases, the former is processed more quickly than the latter, which is simulated by the business processing time. A longer business processing time will simulate a longer I/O operation, such as the reading and writing of a large number of files, database queries, and so forth. Conversely, a shorter business processing time will simulate a shorter I/O operation, such as the accessing of some static web pages and small files.

In the web server implementation, the business processing time is set in the business processing events in the connection class, thereby simulating the requisite time for processing the current business by allowing the system to enter a sleep state.

The business processing time is measured in microseconds and is set at server startup; subsequently, it is tested using Webbench. The business processing time is evaluated over a range of 0 to 1000 microseconds, with the server configured to test at 100 microsecond intervals.

The final result for a web server is that the processing time for each service is proportional to the QPS of the current server. This represents the server's performance in handling the current type of service in a highly concurrent environment. Ultimately, the data values (with one business processing time equating to one QPS) are presented in graphical form using Matlab, thus enabling observation of the trend in the optimal performance of the current web server for different business types in a highly concurrent environment.

B. Experimental results and analysis

1) Single Reactor single-threading web server test

The single Reactor single-threading server was subjected to a series of tests using Webbench, with a concurrency of 20000 and a single HTTP request time of 5 seconds. Table 2 illustrates the business processing time for the test variables.

TABLE II. TABLE TYPE STYLES

Variable	Test range
Number of Sub-Reactors	1-50
Number of threads	1-50

The outcomes of the single Reactor single-threading server examination are illustrated in Figure 20. It can be observed that the QPS of the server declines in conjunction with the expansion of business processing time. The most pronounced decline is evident within the 0－100 range. It is evident that the performance of the single Reactor single-threading Web server is sub-optimal when processing business that requires a significant amount of time. This is due to the fact that all business processing of the single Reactor single-threading Web server is conducted on a single thread. Consequently, if a business process cannot be completed within the allotted time, it will result in the obstruction of other events, thereby preventing other clients from communicating with the server or causing significant delays.

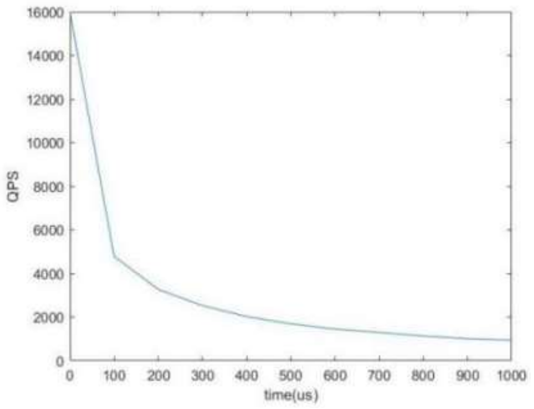


Figure 20. Single Reactor Single Thread Web Server Test Results

2) *Single Reactor Multi-threading Web Server Testing*

The single Reactor multi-threading server was subjected to testing using Webbench with a concurrency of 20000 and a single HTTP request time of 5 seconds. As illustrated in Table 3, the variables under examination are business processing time and the number of threads. The single Reactor multi-threading mode in which the server operates is dependent on the number of threads. By continually adjusting the number of threads, the optimal QPS value that can be attained by this single reactor multi-threading web server with the optimal number of threads is determined.

TABLE III. SCOPE OF TESTING FOR SINGLE REACTOR MULTI-THREADING WEB SERVER

Variable	Test Range
Number of threads	1-30
Transaction processing time (us)	0-1000

Figure 21 illustrates the single Reactor multi-threading web server QPS, which demonstrates a general upward trend from the bottom right to the top left, reaching a peak, before exhibiting a slight decline and subsequently stabilizing. When the value of TIMES (business processing time) is held constant, there is a notable rise in the QPS of the server as the number of THREADS (threads) increases. This evidence substantiates the assertion that multi-threading effectively addresses the issue of performance degradation caused by single thread blocking.

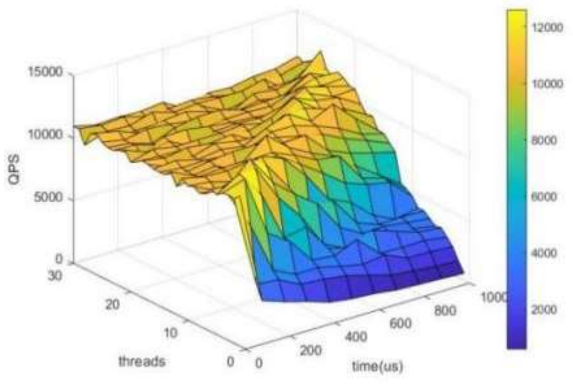


Figure 21. Test Results of Single Reactor Multi thread Web Server

Nevertheless, when a sufficient number of threads are in operation, the server has already reached its optimal performance value. At this juncture, the introduction of additional threads will not yield a positive effect on the optimization of the server. Instead, the frequent switching of threads will result in a reduction in performance.

At this juncture, the optimization component of the thread pool has reached its limit, and the introduction of additional threads does not result in enhanced performance unless the underlying circumstances are altered. The question currently under investigation is what other optimizations are available for single Reactor multi-threading web servers.

3) *Multi-Reactor Multi-threading Web Server Testing*

The Multi-Reactor Multi thread Web Server should be tested with Webbench using a concurrency of 20000 and a single HTTP request time of 5 seconds. In the context of multi-reactor multi-threading web servers, a comparison is drawn between these and single-reactor multi-threading web servers in terms of business processing time and the number of threads. In addition to these two variables, the reactor number is also taken into account, resulting in a new variable. The number of threads and the number of reactors are combined to create a multitude of potential multi-reactor multi-threading web servers, each with its own distinctive characteristics. To ensure comprehensive evaluation, it is essential to assess the performance of each server under varying service processing times.

TABLE IV. MULTI-REACTOR MULTI-THREADING TEST SCOPE UNDER 200US

Variable	Test range
Operational processing time (us)	0-1000

Table 4 illustrates the results of a performance test conducted on a multi-reactor multi-threading web server with a service processing time of 200 microseconds.

The QPS of the web server was obtained from different combinations of the number of Reactors (subReactors) and the number of threads (threads),

measured when processing 200 microseconds of business data. The results are presented in Figure 22.

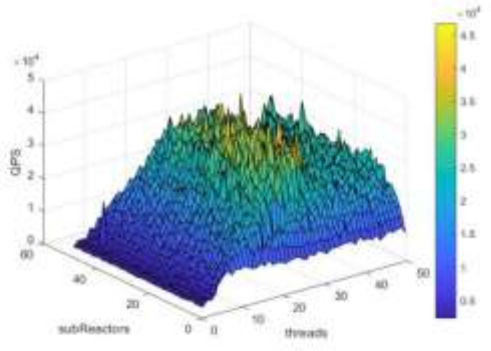
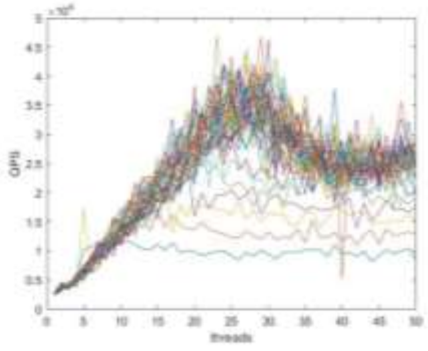


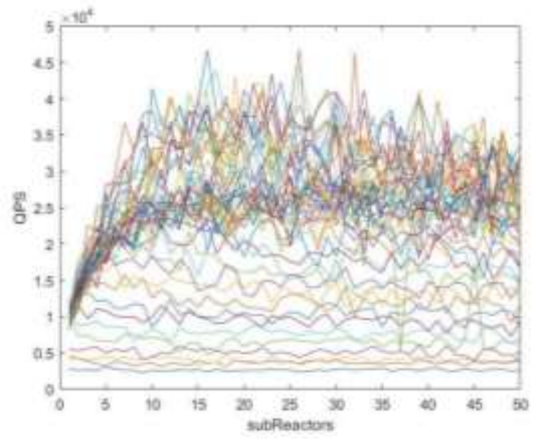
Figure 22. Test Results of Single Reactor Multi thread Web Server

The graph demonstrates a rise in QPS from left to right, reaching a peak and subsequently leveling off. From the outset to the conclusion, the QSP experiences an initial surge and then a decline following a period of sustained peak performance.

As illustrated in Figure 23, the graph is more readily comprehensible when divided vertically into two sections on the X Y axis.：A • m2.”



（a）Y-axis section



（b）X-axis section

Figure 23. Cross-section at 200us

As illustrated in Fig. 23(a), the line graph of QPS with the number of threads is fixed with respect to the number of reactors, with each line in the graph corresponding to a specific number of reactors. It is evident that as the number of threads increases, the QPS value rises and reaches a peak; subsequently, as the number of threads continues to increase, the QPS value declines and then stabilizes. It can be observed that after the optimal QPS has been reached, increasing the number of threads in the thread pool does not result in improved server performance. Instead, it has the opposite effect, leading to a decline in server performance. This is due to the fact that an excess of threads causes meaningless thread switching, which results in the waste of server resources and a subsequent reduction in server performance.

As illustrated in Fig. 23(b), a line graph of QPS versus the number of Reactors is presented for a fixed number of threads, with each line corresponding to a number of threads. It can be observed that as the number of slave reactors increases, the QPS of the server exhibits a marked improvement. Further increases in the number of reactors beyond the optimal point have no discernible impact on performance. The continuation of this process resulted in a decline in performance. A notable increase in performance is observed when the number of Reactors opened on the server is compared to the number of Reactors with a smaller number of threads. This increase reaches its peak immediately and subsequent increases in the number of Reactors have a minimal impact on performance. It can be observed that a specific quantity of slave reactors is sufficient to manage high concurrency. The master reactor distributes client services to the slave reactor at a rate of one per second.

The test results for multi-reactor, multi-threading servers under other business processing times are not significantly different. The primary discrepancy lies in the optimal number of slave reactors and the number of threads required to achieve peak server QPS. The trend of server QPS for these tests is analogous to that observed in the aforementioned test, as illustrated in Figure 24.

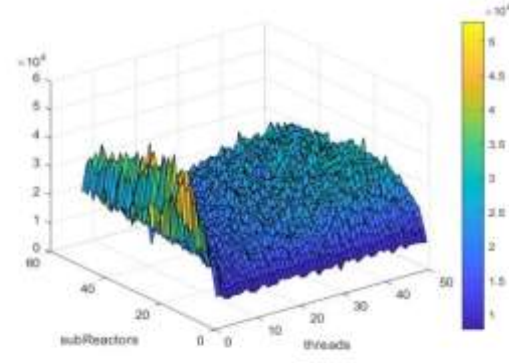


FIGURE (a) Test results at 0us

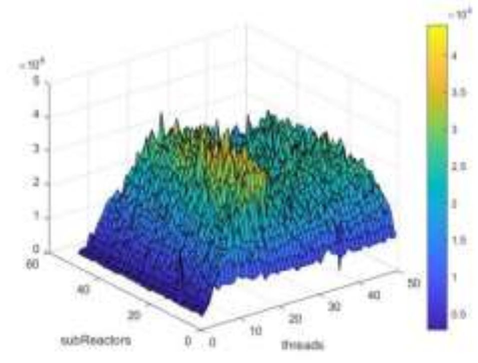


FIGURE (b) Test results at 100us

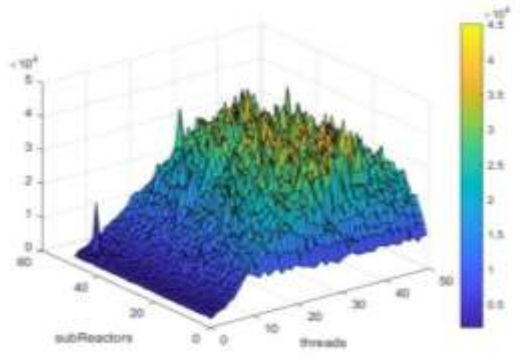


FIGURE (c) Test results at 300us

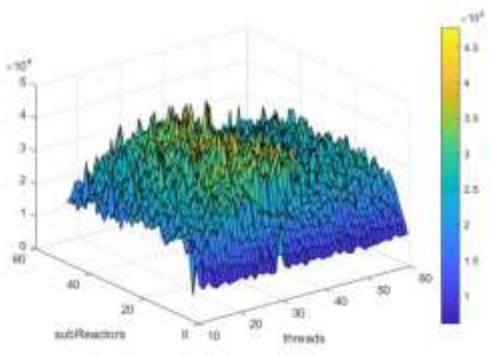


FIGURE (d) Test results at 400us

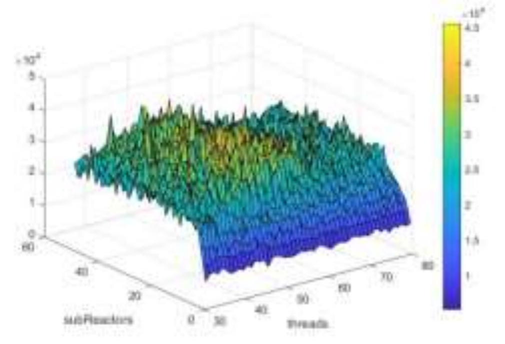


FIGURE (e) Test results at 500us

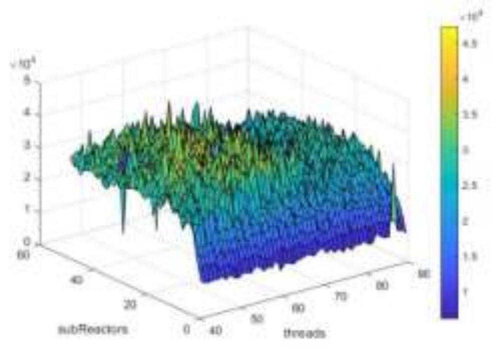


FIGURE (f) Test results at 600us

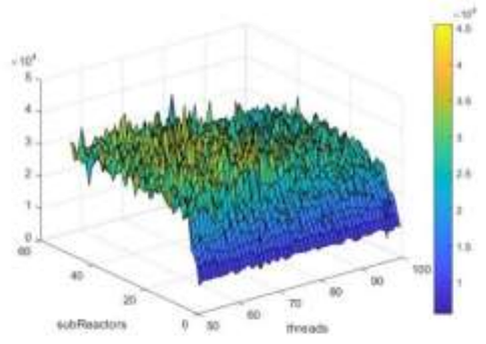


FIGURE （g） Test results at 700us

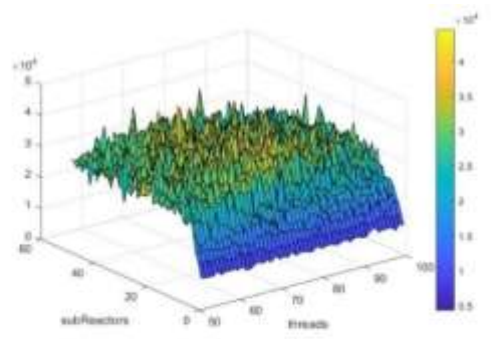


FIGURE （h)Test results at 800us

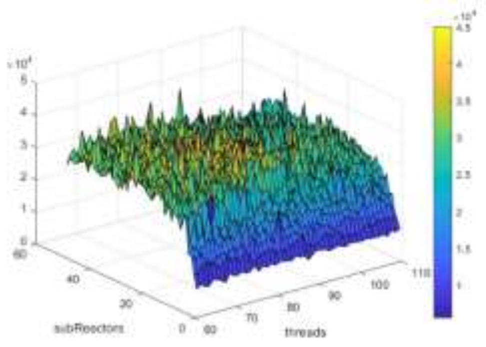


FIGURE （i） Test results at 900us

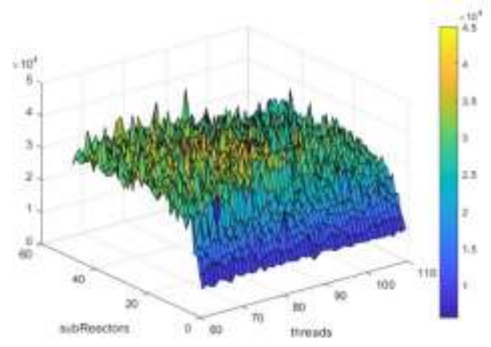


FIGURE （j） Test results at 1000us

Figure 24. Comparison Chart of QPS for Different Time periods

Upon each examination of a server, the source of its performance bottleneck is identified and enhanced, thereby achieving an enhanced performance web server. This process continues until the optimal solution, currently represented by the Multi-Reactor Multi thread Web Server, is reached.

Based on the data shown in the graph, it is evident that the multi - reactor multi - threading web server has significant advantages over the single - reactor single - threading web server. Specifically, the QPS (Queries Per Second) metric of the multi - reactor multi - threading web server is three times higher than that of the single - reactor single - threading web server.

When observing the graph in detail, the single - reactor single - threading web server shows a relatively lower QPS value. Its performance curve rises gradually with the increase of time, but the overall value is not high. This is because, in a single - reactor single - threading architecture, the server can only handle one request at a time. If a new request arrives while the server is processing a previous request, the new request has to wait until the current one is finished. This sequential processing limits the server's throughput and responsiveness, especially when dealing with a large number of concurrent requests.

On the contrary, the multi - reactor multi - threading web server's QPS value shows a remarkable growth. In a multi - reactor multi - threading architecture, multiple reactors are used to monitor different types of events, such as network I/O events. Each reactor can handle multiple threads simultaneously. When a request arrives, it can be quickly assigned to an available thread within the appropriate reactor for processing. This parallel processing mechanism enables the server to handle multiple requests concurrently, greatly improving the processing efficiency.

This substantial improvement in QPS indicates that the multi - reactor multi - threading web server is capable of handling a much larger number of requests simultaneously. In high - concurrency scenarios, where a large number of requests are received in a short period of time, the multi - reactor multi - threading web server can process these requests more efficiently, reducing waiting times and improving overall system performance. The multiple reactors and threads work in parallel, allowing for a greater number of requests to be processed concurrently. Each reactor can handle different sets of requests, and multiple threads within each reactor can execute tasks simultaneously, thereby maximizing the utilization of system resources.

Moreover, the enhanced QPS also implies that the multi - reactor multi - threading web server is better suited for long - term business processing. It can maintain a high level of performance over extended periods, ensuring stable and reliable service delivery. During long - term operation, the single - reactor single - threading web server may encounter bottlenecks due to limited processing capacity, resulting in a decline in performance. For example, if there are a large number of requests queuing up, the single - threaded processing may lead to long - waiting times for some requests, and even cause time - out errors in extreme cases. However, the multi - reactor multi - threading web server can evenly distribute the workload among multiple reactors and threads, avoiding performance degradation caused by long - term operation. This is crucial for applications that require continuous and efficient processing of a large volume of requests, such as e - commerce platforms or large - scale data - intensive services.

In conclusion, the experimental data clearly validates the superiority of the multi - reactor multi - threading web server in terms of high - concurrency handling and long - term business processing capabilities. The significant increase in QPS not only reflects its ability to handle concurrent requests more effectively but also its stability and reliability in long - term operation, making it a preferred choice for web server architectures in high - traffic and data - intensive environments.

The relationship between business processing time and QPS for the final test results of the three Reactor models is presented in a single graph, as illustrated in Figure 25.

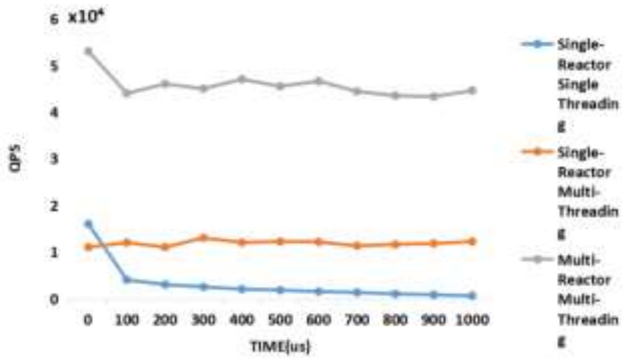


Figure 25. Final Test Results

C. Conclusion of the experiment

The performance of three Reactor models (Single Reactor Single threading, Single Reactor Multi - threading, and Multi - Reactor Multi - threading) in a highly concurrent environment was evaluated, as illustrated in Table 5.

TABLE V. SCOPE OF TESTING FOR SINGLE REACTOR MULTI

Model	Advantages	Disadvantages
Single-Reactor Single Threading	Simple to implement, easy to program and debug; suitable for low concurrency and lightweight business processing scenarios	Poor performance in high concurrency and long time business processing, easy to single thread blocking caused by other requests are delayed processing
Single-Reactor Multi-Threading	The introduction of a thread pooling mechanism serves to enhance the concurrent processing capacity, circumventing the issue of single-thread blocking. This approach is particularly effective in scenarios involving medium concurrency and medium business processing times, offering optimal performance.	The use of multiple threads in a single program can lead to data contention and synchronization problems, necessitating the implementation of locking mechanisms. This, in turn, can result in increased programming complexity and resource overhead. Furthermore, the overhead associated with thread switching may contribute to performance bottlenecks in highly concurrent environments.
Multi-Reactor Multi-Threading	The concurrent processing capability is significantly enhanced by dividing the work among multiple reactors. Each reactor operates independently, reducing competition for resources and improving overall performance. It demonstrates robust performance in high concurrency and long-term business processing.	The complexity of the programming and maintenance processes, coupled with the necessity of dealing with multiple reactor and thread synchronization, gives rise to a considerable challenge in terms of resource management. In order to circumvent performance bottlenecks, it is essential to configure the reactor and thread pool in a reasonable manner.

The Single Reactor Single threading model has a simple structure. However, in a highly concurrent environment, it can only process one request at a time due to having only one thread. This leads to long waiting times as requests queue up, and its performance degrades quickly with increasing concurrency.

The Single Reactor Multi - threading model is an improvement. It uses multiple threads for processing while still relying on a single reactor for event handling. It can handle more concurrent requests compared to the single - threaded model, but the single reactor may become a bottleneck in extremely high - concurrency situations.

The Multi - Reactor Multi - threading model is the most advanced. With multiple reactors and associated threads, it enables a high level of parallelism. Requests can be evenly distributed among the reactors and threads, allowing the system to handle a large number of concurrent requests with short response times. It is highly scalable and suitable for highly concurrent environments like large - scale e - commerce platforms.

These evaluations help in choosing the right architecture for different application scenarios.

V. CONCLUSIONS

In the ever-evolving domain of web server performance, the network programming model, with the Reactor model at its core, has indisputably been a pivotal area of exploration. This comprehensive study has achieved a series of remarkable and significant milestones.

To begin with, it embarked on a detailed investigation into the load capacity of servers that utilize different variants of the Reactor model when confronted with high concurrency scenarios. This entailed meticulously examining how servers with single-reactor single-threading, single-reactor multi-threading, and multi-reactor multi-threading architectures coped under intense traffic loads. Through painstaking research efforts, the study managed to precisely determine the design blueprints and implementation intricacies of high concurrency servers. These were based on the integration of the Reactor model with the highly efficient Epoll mechanism and a well-structured thread pool, all operating within the Linux system environment. The utilization of the Epoll mechanism was particularly crucial as it enabled more efficient event handling and notification, reducing latency and enhancing overall responsiveness.

Subsequently, a series of rigorous stress tests were meticulously carried out on web servers equipped with these diverse Reactor models. The tests were conducted in highly concurrent environments that mimicked real-world, heavy-traffic situations as closely as possible. By subjecting the servers to a barrage of simultaneous requests and analyzing their responses over an extended period, the research team was able to gather a wealth of data. Through painstaking and comprehensive analysis of this data, the final application scenarios for each of the different Reactor models were accurately identified and thoroughly summarized. One of the most significant findings was that, within the constraints and characteristics of the current Linux system, the multi-Reactor multi-threading web server clearly emerged as the preeminent choice for proficiently handling high concurrency requests. It demonstrated superior throughput, shorter response times, and greater scalability compared to its counterparts.

Nevertheless, it is important to acknowledge that, like any research endeavor, this study also has its inherent limitations. The implementation of each model within the scope of this paper was, to some extent, relatively basic and elementary. Crucial elements that are of paramount importance in real-world, practical applications, such as advanced load balancing techniques and sophisticated caching mechanisms, were not given the full consideration they deserved. These factors can have a profound impact on the overall performance and user experience of a web server. For instance, without proper load balancing, servers may experience uneven workload distribution, leading to bottlenecks in some areas while other resources remain underutilized. Similarly, an effective caching mechanism can significantly reduce the need to repeatedly process the same data, thereby enhancing efficiency.

Moreover, the testing regime employed had its own set of constraints. The testing scope was somewhat narrow, primarily focusing on a specific range of concurrency levels and business processing time frames. This meant that it failed to comprehensively encompass all the potential and diverse business scenarios and load conditions that a web server might encounter in the real world. There could be niche applications or extreme traffic spikes that were not adequately represented in the tests, potentially leading to inaccurate generalizations about the performance of the models.

Looking to the future, there is a vast expanse of opportunities and areas ripe for further exploration and research. Future work will be centered around optimizing the design of the multi-reactor model. This will involve delving deep into exploring more efficient thread management strategies. For example, investigating techniques to minimize thread context switching overheads, which can consume significant computational resources and degrade performance. Additionally, the development of more advanced load balancing strategies will be a key focus. This could involve dynamic load balancing algorithms that can adapt to changing traffic patterns and server loads in real-time, ensuring optimal resource utilization.

The model will also be extended to a broader array of real-world applications. In particular, scenarios involving dynamic content processing, such as real-time video streaming or interactive web applications, and large-scale data transmission, like bulk file transfers or database synchronization, will be explored. By applying the model to these diverse settings, its adaptability and effectiveness can be accurately gauged.

Furthermore, it is highly recommended that performance evaluations be carried out across a wide variety of test environments. This would involve incorporating different hardware configurations, ranging from low-end consumer-grade systems to high-performance enterprise servers. Different network conditions, such as high-latency satellite connections or low-bandwidth mobile networks, as well as diverse load modes, including bursty traffic patterns and steady-state loads, should all be considered. By doing so, a more exhaustive and accurate set of performance data can be collected, providing a more comprehensive understanding of the model's capabilities.

Finally, further efforts will be dedicated to developing more efficient synchronization mechanisms. In multi-threading environments, resource contention and excessive thread switching can be major bottlenecks. By devising more intelligent synchronization strategies, such as lock-free data structures or fine-grained locking techniques, the aim is to minimize these overheads and thereby bolster the server's concurrent processing capabilities to new heights. This will ensure that web servers based on the multi-reactor model can meet the ever-increasing demands of modern, high-traffic, and data-intensive applications.

ACKNOWLEDGMENT

The work presented in this paper was supported in part by the Key Industry Chain Technology Research Project of Xi'an Science and Technology Bureau (Grant No. 23ZDCYJSGG0018-2023), the Science and Technology Planning Project (Grant No. GX2412) and the Research project on teaching reform of education in Shaanxi province (Grant No. 23BY082).

REFERENCES

[1] Yuguang Zhang, a layered architecture system for high concurrent processing integrating different scenarios [J]. Communication Technology, 2020, 53(01): 93-100.

[2] Chen R, Mou Y, Li W. A provably secure multi-server authentication scheme based on Chebyshev chaotic map [J]. Journal of Information Security and Applications, 2024, 83: 103788.

[3] Babou N, Boudhar M, Rebaine D. Two-machine job shop problem with a single server and sequence-independent non-anticipatory set-up times [J]. Discrete Optimization, 2024, 53: 100845.

[4] Li YJ, Tian F, Ni ZY. Server architecture design for highly concurrent complex civil aviation services [J]. Computer Applications and Software, 2016, 33(05): 4-7, 39.

[5] Li Jianhua, Xia Flood, Luo Mingquan. Research and implementation of high concurrency WeChat public development based on ThinkPHP and Redis [J]. Computer Application and Software, 2019, 36(02): 108-112.

[6] Yuntao Xu, Wujun Xu, Menglin Zhai. A high concurrency iris recognition system based on B/S

architecture [J]. Computer Engineering, 2019, 45(08): 102-106, 112.

[7] Junfeng Li, Mingxin He. Design and implementation of high concurrency Web airline ticket spike system [J]. Computer Engineering and Design, 2013, 34(03): 778-782.

[8] Wang Jiye, Ding Weilong, Gao Lingchao et al. A sensory data access service supporting high concurrency [J]. Small Microcomputer Systems, 2017, 38(12): 2703-2706.

[9] Yannan Wang, Huarui Wu, Feng Huang. Performance optimisation analysis and research on high concurrency web application system [J]. Computer Engineering and Design, 2014, 35(08): 2976-2981.

[10] Li KW. Practice of high concurrency technology architecture in the Internet [J]. Digital Communication World, 2019(03): 65-66.

[11] Jiexin Zhang, Jianmin Pang, Zheng Zhang, et al. An approach to quantify service quality of mimetic constructed web servers [J]. Computer Science, 2019, 46(11):109-118.

[12] Jiexin Zhang, Jianmin Pang, Zheng Zhang. A method for quantifying web server heterogeneity by mimetic construction [J]. Journal of Software, 2020, 31(2):564-577.

[13] Sun B,Sun M. Concurrency and Operating Systems, Processors, and Programming Languages [J]. Highlights in Science, Engineering and Technology, 2023, 39: 881-887.

[14] Roper M D, Ishihara T, Olsson R A. Critical Performance Factors in Web Server Design: Experience Implementing CoW, a Cooperative Multithreading Web Server [J].

[15] Moslehian A S. An Experimental Integration of io uring and Tokio: An Asynchronous Runtime for Rust [D].

[16] Karsten M, Barghi S. User-level threading: have your cake and eat it too [J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2020, 4(1): 1-30.

[17] Schmaus F, Fischer F, Hönig T, et al. Modern Concurrency Platforms Require Modern System-Call Techniques [J]. 2021.

[18] Ma C, Chi Y. Evaluation test and improvement of load balancing algorithms of Nginx [J]. IEEE Access, 2022, 10: 14311-14324.

[19] Li G, Li J. Optimising low-power task scheduling for multiple users and servers in mobile edge computing by the MUMS framework [J]. Heliyon, 2024, 10(11).

[20] Liu T. Software design of wireless adaptation terminal server in distributed testing system [D]. University of Electronic Science and Technology, 2023. doi: 10.27005/d.cnki.gdzku.2023.000535.

[21] Nie Fanjie. Research and case analysis of high performance server-side framework technology based on Reactor pattern [D]. Zhejiang University of Technology, 2020. doi: 10.27786/d.cnki.gzjlg.2020.000175.

[22] E Qin. Research and improvement of load balancing algorithm based on Nginx high concurrency server [D]. Wuhan University of Technology, 2020. doi: 10.27381/d.cnki.gwlgu.2020.000368.

[23] Ge Y., Li H. Ochre, Li S. Fei. An adaptive dynamic load balancing design and implementation for web server cluster [J]. Computer and Digital Engineering, 2020, 48(12):3002-3007.

[24] Wu Chen. Research and improvement of server cluster load balancing strategy based on Nginx [D]. South China University of Technology, 2020. doi:10.