



湖南工程學院

# 毕 业 设 计

题 目： 高并发反向代理服务器的设计与实现

学 院： 计算机与通信学院

专 业： 计算机科学与技术 班级： 计算机 1302

学 号： 201303010202

学生姓名： 丁 涛

导师姓名： 彭 梦

完成日期： 2017 年 6 月 5 日

# 诚 信 声 明

本人声明：

1、本人所呈交的毕业设计（论文）是在老师指导下进行的研究工作及取得的研究成果；

2、据查证，除了文中特别加以标注和致谢的地方外，毕业设计（论文）中不包含其他人已经公开发表过的研究成果，也不包含为获得其他教育机构的学位而使用过的材料；

3、我承诺，本人提交的毕业设计（论文）中的所有内容均真实、可信。

作者签名：

日期：2017 年 6 月 5 日

# 湖南工程學院

## 毕业设计（论文）任务书



题目：高并发反向代理服务器的设计与实现

姓名 丁涛 学院 计算机与通信学院 专业 计算机科学与技术 班级 计算机 1302 学号 201303010202

指导老师 彭梦 职称 讲师 教研室主任 李珍辉

### 一、基本任务及要求：

#### 1 设计任务特点及分析

高并发反向代理服务器的设计与实现主要是使用 Libevent 作为异步网络事件库，并在类 UNIX 系统下运行服务级别软件，为了良好的可维护性及可移植性，软件使用纯 C 语言编写。根据 C 语言的特性，也满足高性能这一项需求。为了追求极致的性能，本软件在设计上应该尽可能使用异步 IO，所有 IO 均设置为非阻塞方式读写。Libevent 作为跨平台开源网络事件库，也支持对文件的异步操作，封装级别较高，特别适合应用层的开发。

#### 2 主要任务

- (1) 日志模块：本模块主要功能为异步记录程序执行的信息至文件，方便后期维护 DEBUG。
- (2) 代理模块：本模块主要实现异步请求及响应的代理转发。
- (3) 缓存模块：本模块主要实现特定响应信息写入文件的操作，同时判断请求是否已缓存。

### 二、进度安排及完成时间：

2017 年 1 月 5 日-2017 年 3 月 10 日 明确课题任务及要求，搜集课题所需资料，了解本课题研究现状、存在问题及研究的实际意义；写好选题报告和文献综述。

2017 年 3 月 11 日-2017 年 3 月 28 日 毕业实习。

2017 年 3 月 29 日-2017 年 4 月 10 日 查阅相关资料，自学相关内容，确定软件总体设计方案。

2017 年 4 月 11 日-2017 年 4 月 20 日 各部分模块设计及部分调试，系统总体联调。

2017 年 4 月 21 日-2017 年 5 月 21 日 整理资料，撰写毕业设计论文。

2017 年 5 月 21 日-2017 年 6 月 3 日 毕业论文审定、打印，答辩准备。

2017 年 6 月 6 日-2017 年 6 月 8 日 答辩。

# Content

<b>Abstract .....</b>	<b>1</b>
<b>Preface .....</b>	<b>3</b>
<b>1 Introduction.....</b>	<b>4</b>
1.1    The significance of the study .....	4
1.2    The purpose of the study.....	4
1.3    Research status in China and abroad .....	4
<b>2 Hypertext Transfer Protocol .....</b>	<b>6</b>
2.1    History of HTTP .....	6
2.2    HTTP/1.1 .....	7
2.2.1    Start-line .....	7
2.2.2    Header-fields .....	7
2.2.3    Examples .....	8
2.3    Summary.....	8
<b>3 Operational principle of the reverse proxy server.....</b>	<b>9</b>
3.1    Content delivery network .....	9
3.2    Intelligent DNS resolution.....	9
3.3    HTTP Cache.....	10
<b>4 UNIX IO Mode.....</b>	<b>11</b>
4.1    Blocking and Non-Blocking IO .....	11
4.2    Synchronous and Asynchronous IO.....	11
4.3    Libevent .....	11
<b>5 Detail design .....</b>	<b>13</b>
5.1    Log system module.....	13
5.1.1    The necessity of the existence of the log system .....	13
5.2    Design of log system .....	14
5.2.1    General log system .....	14
5.2.2    Improvement of log system.....	14
5.2.3    Improvement design .....	14
5.3    Proxy module and cache module .....	17
5.3.1    Overall design of proxy and cache .....	18
5.3.2    Saving the response into cache file .....	19

<b>6 Implementation</b>	21
6.1 Main function implementation	21
6.2 Configuration file resolution	21
6.3 Log system implementation	21
6.3.1 Log module initialization	21
6.3.2 Write the log	22
6.3.3 Asynchronous mechanism	22
6.4 Proxy and cache implementation	22
6.4.1 File cache implementation	23
6.4.2 Cache hits	23
6.4.3 Cache misses	24
<b>7 Software test and comparison</b>	25
7.1 Test environment	25
7.1.1 Hardware and software environment	25
7.1.2 Network topology	25
7.2 Test environment configuration	25
7.3 Test method	26
7.3.1 Access the un-cached static html file	26
7.3.2 Access the cached static html file	26
7.3.3 Access the un-cached picture	27
7.3.4 Access the cached picture	28
7.4 Comparison	29
7.4.1 Comparison method	29
7.4.2 Comparison result	29
<b>8 Summary</b>	31
<b>Bibliography</b>	32
<b>Acknowledgements</b>	34
<b>Appendix</b>	35
Program user's manual	35
Config	35
Comment	35
Log	35

Cache .....	35
Original web server's address .....	35
Source Code .....	36
Evaluation data .....	54
Raw data of test .....	54
Test program source code .....	54

# THE DESIGN AND IMPLEMENTATION OF HIGH CONCURRENT REVERSE PROXY SERVER

**Abstract:** Reverse proxy server is a middle server with caching function of the server and the client. In the high concurrent environment, the reverse proxy server with caching function can effectively reduce the load of the source server and reduce the amount of network bandwidth of the source server. This paper first briefly introduces the HTTP protocol, the reverse proxy server and the UNIX's IO model, designed and implemented a high concurrent reverse proxy server of high performance with caching function named ktrix based on Libevent cross-platform event-driven network library under UNIX environment. In order to test the performance of the software, this paper uses nginx as a comparison object, simulates a high concurrent environment by software and tests ktrix and nginx in the same environment. Test results shows that ktrix in high concurrent environment performance than nginx about 10% higher, the software overall to achieve the desired design goals.

**Keyword:** High Concurrent; Reverse Proxy; Proxy Caching; Libevent; Ktrix

## 高并发反向代理服务器的设计与实现

**摘 要：**反向代理服务器是介于源服务器与客户端之间，具有缓存功能的中间服务器。在高并发环境下，具有缓存功能的反向代理服务器能有效降低源服务器的负载，同时减少源服务器对网络带宽的占用量。本文首先对 HTTP 协议、反向代理服务器的工作方式和 UNIX 的 IO 模型做简要的介绍，以 UNIX 系统为平台，使用 Libevent 跨平台事件驱动网络库设计并实现了一款基于事件驱动模型的高性能、高并发反向代理缓存服务器 ktrix。为了测试软件的性能，本文使用 nginx 作为比较对象，通过软件模拟出高并发环境并测试相同环境下的 ktrix 及 nginx。测试对比结果显示，ktrix 在高并发环境下的性能比 nginx 高大约 10%左右，软件总体上达到了预期的设计目标。

**关键字：**高并发；反向代理；代理缓存；Libevent；Ktrix



## Preface

HTTP protocol as the Internet's most popular application layers protocol, has occupied the main traffic packet in the Internet traffic forwarding. A single web server when receives a large number of HTTP connection requests, will inevitably lead to single Web server response slower than usual status. Restricted by server bandwidth, a large number of advisedly dropping packets will increase the data response time; advisedly dropping packets will also affect the router and switch on the connected link, increase its calculation pressure and further affect the response time of the Web request. The reverse proxy server also receives a large number of requests from the client, so it is also important to speed up the response from a high concurrent environment.

This paper gives a brief description of the overall framework of the distributed HTTP server, focusing on the distributed framework of the reverse proxy server. Then analyze the asynchronous and non-blocking network IO mode which is suitable for high concurrent environment. Then designed and implemented a reverse proxy server based on fast portable asynchronous network library Libevent, and the test its performance compared to nginx. The test result verifies the feasibility and effectiveness of this project.

# 1 Introduction

## 1.1 The significance of the study

The client initiates the request for the server at the far location in the geographical position, and transfers from the multi-level switch and router, occupying the network bandwidth and the computing power of the equipment. Geographically, several back-to-back servers with caching capabilities are placed closer to the client, which will significantly to reduce the computational pressure on the intermediate network device and reduce the latency between the client and the server. When a large number of users access server's resources at the same time, the layout of the reverse proxy server can respond to the pressure on the response to each buffer with the back of the server, can speed up the source server to the client request calculation speed.

## 1.2 The purpose of the study

The nginx reverse proxy server is designed primarily to reduce the computational effort of the source server and to proxy the requests sent by the client in a load-balanced manner. But in response to the contents of the cache design is not satisfactory. The goal of this project is to emphasize the intermediate result cache, reduce the bandwidth occupied by the middle network, and reduce the delay of the client to receive the response, so as to improve the user experience.

## 1.3 Research status in China and abroad

Abroad research focus on the design of HTTP intermediate cache software is more, well-known squid. Squid is a caching function of the Web proxy server, support HTTP, HTTPS, FTP and other common protocol agent. It reduces the bandwidth usage and latency between the client and the server by responding to the most frequently accessed pages of the cache. Its configuration is more complex, the design of the pursuit of universal, does not apply to the complex production environment.

There are two domestic commercial software for the domestic design and development of the cache server, the morequick (秒开缓存) and webcache (缓存大师). Seconds to open the cache and cache master as closed source commercial software, its underlying implementation principle and logic does not have a corresponding paper or monograph to

describe it. These two software-oriented groups for the IPS, Internet cafes, etc., the user is the proxy cache. Their cache proxy principle is by the switch port mirroring function to copy the gateway export packet to the server, the cache server by detecting the client to send the request, disguised as the source server to send HTTP 302 redirect, redirect to the machine, force the client Access the cache server, in order to achieve the purpose of saving bandwidth.

## 2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, and hypermedia information systems.<sup>[5]</sup> HTTP is the foundation of data communication for the World Wide Web.

Development of HTTP was initiated by *Tim Berners-Lee* at CERN in 1989. Standards development of HTTP was coordinated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), culminating in the publication of a series of Requests for Comments (RFCs). The first definition of HTTP/1.1, the version of HTTP in common use, occurred in RFC 2068 <sup>[4]</sup> in 1997, although this was obsolete by RFC 2616 <sup>[5]</sup> in 1999 and then again by RFC 7230 <sup>[6]</sup> and family in 2014.

A later version, the successor HTTP/2, was standardized in 2015, and is now supported by major web servers.

### 2.1 History of HTTP

The original version of HTTP called *HTTP/0.9* <sup>[2]</sup> has no standard specification and be abandoned for decades. Subsequently with the extensive use of HTTP, the Internet Engineering Task Force (IETF) has developed standard RFC1954 to *HTTP/1.0* <sup>[3]</sup> in May 1996 which is the first standard version of HTTP. Soon after with the widely use of HTTP, the weakness of HTTP/1.0 had come to light. After finish data transfer, the old version of HTTP, HTTP/1.0 and under, will disconnect current TCP connection, which will cause redundancy TCP connection establish between client and server while the client may will revisit another page on the same website. Therefor the IETF has developed another standard RFC2616 to *HTTP/1.1* <sup>[5]</sup> in June 1999 which is the second version of HTTP.

In June 2014, HTTP/1.1 has been updated by RFC7230 <sup>[6]</sup>, RFC7231 <sup>[7]</sup>, RFC7232 <sup>[8]</sup>, RFC7233 <sup>[9]</sup>, RFC7234 <sup>[9]</sup> and RFC7235 <sup>[10]</sup>. All those RFC documents are focus on the interpretation of the original semantic fuzzy part, make the specification that is more understandable and easy to read.

HTTP/1.1 uses strings to control transmission rather than bit control, which leads to a large amount of redundant information in the HTTP/1.1 message. Under to improve HTTP/1.1 circumstances, HTTP has developed newest standard RFC7540 to *HTTP/2* <sup>[12]</sup> in May 2015. HTTP/2 enables a more efficient use of network resources and a reduced perception of latency

by introducing header field compression and allowing multiple concurrent exchanges on the same connection <sup>[12]</sup>.

## 2.2 HTTP/1.1

HTTP/1.1 message consist of *start-line*, *header-field* and *message-body*. Each part of them is separated by one or more *CRLF*. The *CRLF* is string “\r\n” as well as ASCII code 0xDA <sup>[5]</sup>. The syntax of HTTP/1.1 as shown in Figure 2-1.

```
<start-line> CRLF
<header-fields> CRLF
CRLF
<message-body>
```

Figure 2-1 Syntax of HTTP/1.1 message

Message *header-field* and *message-body* are optional in HTTP/1.1.

### 2.2.1 Start-line

HTTP/1.1 message can be divided into two types, *request message* and *response message* <sup>[2]</sup>.

The start-line’s syntax of request message shows as follows.

```
<method> <request-URI> <version>
```

The start-line’s syntax of response message shows as follows.

```
<version> <status> <reason-phrase>
```

The *method* represents the client wants the server to perform the action of the resource, such as *GET*, *POST* etc. The *request-URI* represents a Uniform Resource Identifier. The *version* represents current version of HTTP.

### 2.2.2 Header-fields

Header-field is series of key-value pairs. A HTTP/1.1 message can carry zero or more header-fields in order to control HTTP. Each header-field separated by *CRLF* (see 2.2). The syntax of header-field shows as follows <sup>[6]</sup>.

```
<header-name>: <header-value>
```

### 2.2.3 Examples

A typical HTTP/1.1 message for both client and server are as shown in the figure 1.1.

HTTP/1.1 request message	HTTP/1.1 response message
GET /index.html HTTP/1.1 Host: www.hnie.edu.cn Accept: html, q=0.8; *; Connection: keepalive	HTTP/1.1 200 OK Content-Type: text/html Content-Length: 12345  <html>...</html>
POST /query.cgi HTTP/1.1 Host: jwc.hnie.edu.cn Content-Type: text/plain Content-Length: 30  This is a POST method example.	HTTP/1.1 301 Moved Permanently Location: http://cdn.hnie.edu.cn/ Content-Type: text/plain Content-Length: 36  Please go to http://cdn.hnie.edu.cn/

Figure 2-2 Typical HTTP/1.1 message

## 2.3 Summary

HTTP is designed to allow the intermediate cache server to improve the quality of communication between the client and the server. High traffic sites typically benefit from the load balancing capabilities of the distributed Web cache server upstream server to improve response time. The Web browser accesses the Web resources of the Web cache server before accessing the source server and, if possible, reusing them to reduce network traffic. The HTTP proxy server within the private network can achieve the purpose of communicating with the client by relaying HTTP messages by using an external server as a proxy server without the need for multiple public network routable addresses.

## **3 Operational principle of the reverse proxy server**

### **3.1 Content delivery network**

A content delivery network or content distribution network (CDN) is a globally distributed network of proxy servers deployed in multiple data centers. The goal of a CDN is to serve content to end-users with high availability and high performance. CDNs serve a large fraction of the Internet content today, including web objects (text, graphics and scripts), downloadable objects (media files, software, documents), applications (e-commerce, portals), live streaming media, on-demand streaming media, and social networks.

The term CDN means many things to different people and is an umbrella term that covers a lot of different types of content delivery services. Video streaming, software downloads, web and mobile content acceleration, licensed/managed CDN, transparent caching, and services to measure CDN performance, load balancing, multi-CDN switching and analytics and cloud intelligence. It's a complex ecosystem with a lot of vendors both large and small and some CDN vendors cross over into other industries like security and WAN optimization.

Content owners such as media companies and e-commerce vendors pay CDN operators to deliver their content to their end-users. In turn, a CDN pays ISPs, carriers, and network operators for hosting its servers in their data centers.

### **3.2 Intelligent DNS resolution**

Ordinary DNS server is only responsible for the user to resolve IP records, rather than to determine where the user from, this will cause all users can only be resolved to a fixed IP address. Intelligent DNS resolution flowchart smart DNS overrides this concept. Intelligent DNS will determine the user's way, and make some intelligent processing, and then the intelligent judgment of the IP back to the user. For example, DNSPod intelligent DNS will automatically determine the user's ISP is China Telecom or CERNET, and then intelligent return to China Telecom or CERNET server IP.

To provide CDN services, intelligent DNS, according to the user's source, by city or region to determine the user source, in essence, multi-line multi-zone intelligent DNS, updated, such as 8GDNS multi-line multi-area intelligent DNS, can automatically determine the user's ISP is Shanghai Telecom or Guangdong Telecom, and then intelligent return to the corresponding

Shanghai Telecom and Guangdong Telecom server IP. At the same time a new generation of intelligent DNS such as 8G DNS there are downtime detection function, if the server downtime, a new generation of intelligent DNS real-time detection of downtime server IP, and DNS resolution request to the normal operation of the server. Thus, providing a highly reliable service without downtime.

### **3.3 HTTP Cache**

The HTTP protocol defines several headers for intermediate cache servers to make cache decisions, the most important of which is the Cache-Control header. As the RFC2616<sup>[5]</sup> published in 1999 has been abolished, this paper will be based on the latest RFC7234<sup>[10]</sup> described Cache-related standards, the preparation of the program.

According to RFC7234<sup>[10]</sup> definition, related to the cache-related head are: Age, Cache-Control, Expires, Pragma, Warning. The program will not be cached clearly defined should not be cached response, the other are passed to the cache module for the cache module to determine whether to cache the current response. The definitions of these headers are inseparable from the conditional request headers defined in RFC 7232<sup>[8]</sup>, and the program will be coded according to the definition of the two.

In particular, similar to the Set-Cookie and other non-standard head, the program will do some compatibility processing. If the response to the design of the client to set the Cookie, even if the other header information should be cached the current response, the proxy cache server will not be cached.



## 4 UNIX IO Mode

### 4.1 Blocking and Non-Blocking IO

Blocking IO and non-blocking IO concerns the state of the current process (or thread) waiting for IO to call the result (or message). Blocking IO refers to when the caller initiates an IO request. The current process (thread) will be hung before entering the IO result and the non-executable state will be entered until the called function returns the IO result to resume execution into the ready or operational state. Non-blocking IO refers to the caller to initiate an IO request, the call function immediately return, there may be returned IO results, there may be returned error code, then the caller can choose a certain time to start the IO request again until to the expected IO result; during this time, the caller can execute other functions and will not be suspended by the system.

### 4.2 Synchronous and Asynchronous IO

Synchronization and asynchronous attention to the function (process or thread) between the message transfer mechanism. Synchronous call, that is, when the caller makes a call request, the call will not be returned until the result is fetched. Once the call returns, then the result is retrieved. Therefore, the caller is actively waiting for the current call result. Asynchronous calls, in contrast to synchronous calls, are returned directly after the call is issued, but no results are returned. Thus, when an asynchronous invocation process is issued, the caller does not immediately get the result of the call, but after the call is made, the caller is informed by the system by sending a status, signal, notification, etc., or by executing a registered callback function (Hook function) to handle the current return value.

### 4.3 Libevent

Different operating systems are asynchronous and non-blocking support and API interface are different, such as POSIX select, Windows IOCP, Linux epoll and so on. In the preparation of cross-platform requirements of the high concurrent procedures, usually need to adapt to different platforms IO interface functions to achieve the purpose of cross-platform software.

Libevent as an open source network library, encapsulated the asynchronous interface of different platforms at the bottom of the interface, provides a simple interface to the registration

callback function based on the event API, is a very suitable for cross-platform requirements of the development of the network library. Libevent itself is written in C language, the efficiency is very high, in the design support events can be extended, focus on network IO optimization, support for multi-threaded access. In cross-platform, through the conditions of compilation, to support Windows, Unix-like and other operating systems.

Bufferevent is the event structure with the caching function in the libevent library. The application layer notifies libevent of the function address that should be called when the read and write event occurs by creating the bufferevent structure and registering the corresponding event callback function. Libevent by checking the bufferevent read and write the size of the cache to determine whether the upper call should call the callback function, which will read and write IO read and write asynchronous non-blocking IO model.

## **5 Detail design**

### **5.1 Log system module**

#### **5.1.1 The necessity of the existence of the log system**

##### **Collect run records and debug information.**

In many cases, there are a lot of logic in the development of applications need to analyze debugging to ensure that the program running logic is in line with requirements and rules. In the unit testing phase, the program context information can be obtained by outputting information directly by printing directly. But to analyze a system running information and debugging process, whether it is unit test or tracking debugging can't meet the demand, so this time need to log down, and then through appropriate means to monitor the output of the log, so the program runs record at a glance. This is the most common log for logging application startup, and it is easy to see the execution of the application through the log.

##### **Collect application warnings, errors and crash messages.**

Even if the perfect system will have loopholes, when the program runs in the production environment, can't guarantee that each application can be perfect for the normal operation of the perennial, in the course of running, will certainly encounter a lot of problems, these problems may be due to the environment, network, security vulnerabilities caused by, but in the development of the situation can't be encountered. So, this time you need to record the wrong information, through the analysis of the log, locate the problem and repair them.

##### **Collect and analyze application performance.**

For large web applications, slowing a second means losing a few percentage points of the user, so solving the performance problem will save huge benefits for the site. At this point the optimization performance will depend on the site's log system, especially the subdivision of the slow log, such as the application which has a set of steps to complete a page output, they are the basic framework, connection cache, connect the database, template rendering, In each step of the time-consuming records, and finally recorded in the log inside, in the future analysis of the time to check the longest time each step of the log, according to this log to do targeted

performance optimization.

## **5.2 Design of log system**

### **5.2.1 General log system**

Most of the software log system are use of synchronous to writing log in the very beginning design of the software. The advantage of this design is to simplify the complexity of the log system, enhance software maintainability; the disadvantage is that the software will waste very much computing resources, because the CPU had to wait for the log information to write the file before continue to execute.

Nginx as an asynchronous Web server, its log system is support the UDP communication, to send the log information to a dedicated log server which is responsible for writing the log and save to file. The way of use of network communication to write the log, although the log can be sent asynchronously, may occur packet loss, resulting in log information is lost. Moreover, the use of network transmission log will increase the network card load, IP package assembly/disassembly and other network protocol processing will consume a lot of CPU resources.

### **5.2.2 Improvement of log system**

This thesis proposed new design of asynchronous log write mode. In this mode, the program can continue to perform other tasks when give the log information which it need to save. The log module gets the log information that needs to be written in the log, and set the log write timeout processing function, in the detection of log write failure repeatedly add write log events to ensure that log information is never lost.

### **5.2.3 Improvement design**

If the file descriptor is temporarily unavailable when the file IO is temporarily unavailable, the program will go into the pending state until the file can be written, and then the program status is restored. Then, the log information will be written when the program is running. Waiting for IO and make the CPU into the idle state, will greatly reduce the concurrent throughput of the program.

Therefore, this program used in the design of asynchronous non-blocking IO, other

functions if we need to print log information, the log as an event, register write event callback, by the time the Libevent notification log system file descriptor is ready, the program can write log in to the file. The overall process is illustrated in Figure 5-1.

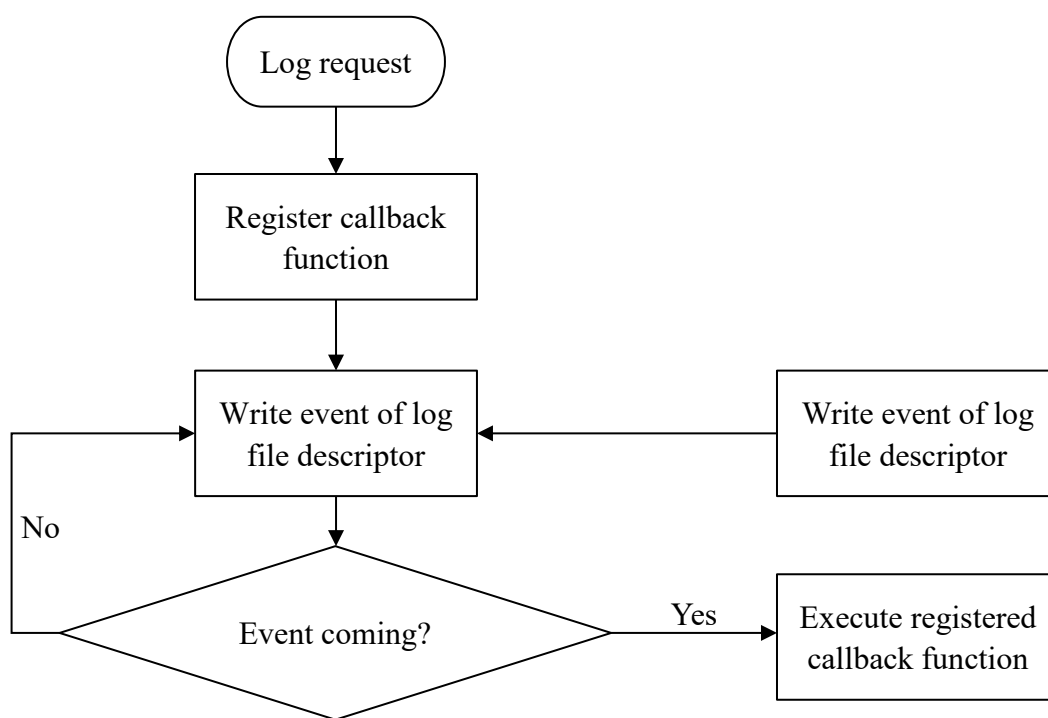


Figure 5-1 Flow diagram of log system

## Event timeout and write synchronization

When a program is interrupted by some unpredictable signal, it may cause the registered write event to be lost. The long-running program will cause the underlying wait for the event to be written too much and take up a lot of memory, so it needs to execute the write event Regular reboot to ensure that the current write event must be awakened.

Asynchronous writes may be interrupted by other identical log writes, causing the event information of the write file to be out of sync, so the file write lock protection is required. Considering that the log write will only write to the file, the program uses a mutex instead of a read and write lock on the lock selection.

In the presence of the above two restrictions, the final log module design workflow as shown in Figure 5-2.

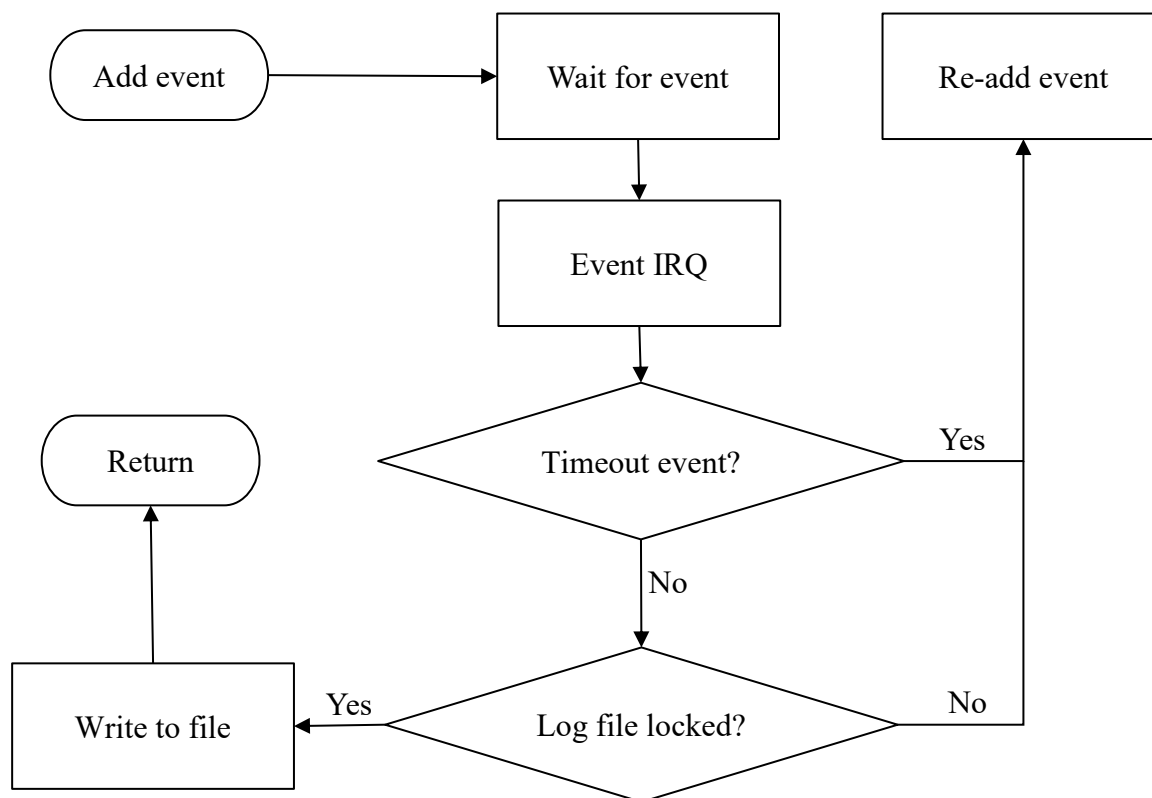


Figure 5-2 Event handle of log system

### Asynchronous log key pseudo-code

```

void log_cb(int fd, short what, void * ctx)
{
    char *log_str = ctx; /* Log information */
    if ( time_out || lock_the_log_file() == false ) {
        /* Time out detected, or current file is busy. Re-add log event. */
        event_base_once(g_event_base,  g_log_fd,  EV_WRITE,  log_cb,  log_str,
        &g_time_out);
        return ; }
    /* At this point, we successfully lock the log file */
    write(fd, log_str, log_str_len);
    unlock_the_log_file();
    free(log_str);
    return ; }

void log(log_level_t level,

```

```

    const char * file,
    const char * func,
    const char * line,
    const cahr * format,
    ...) {
/**<
 *   When this func runs out, all variables in stack will
 *   lost, so we decided to set log_str in heap to avoid
 *   out of range pointer access.
 */
char * log_str = malloc(LOG_STRING_DEFAULT_SIZE);
/* Variable argument function call, make a log string to write. */
sprintf(log_str, "[%d][%s][%s][%d] ...", level, file, func, line, format);
/* Set log callback function when log file descreptor redy to write. */
event_base_once(g_event_base,    g_log_fd,    EV_WRITE,    log_cb,    log_str,
&g_time_out);
return ; }

```

### 5.3 Proxy module and cache module

Mainstream proxy server in market start HTTP packet resolution immediately after receives the client initiated request. In the resolution of incomplete messages, at the same time, determine whether there is a local cache. If it is checked that there is no cache file in local file system, the received request is forwarded to the source server, even if a complete request has not been received. This design can speed up the message forwarding speed, reduce the response time.

When the link quality is so poor between the client and the source server which will course often lose packets. While parsing and forwarding the client's request when lost packet occurs will let the retransmission pressure on the source server. As shown as Figure 5-3, assuming that the client only sends an incomplete HTTP request (either due to poor link quality or client interruption of the connection, etc.), the proxy server forwards the incomplete request to the server, and the proxy server waits for the client to send the remaining request data, the source

server also needs to wait for the proxy server to forward the remaining request data during this time. This will increase the load on the source server, and vulnerable to DDoS attacks, lead the source server to downtime.

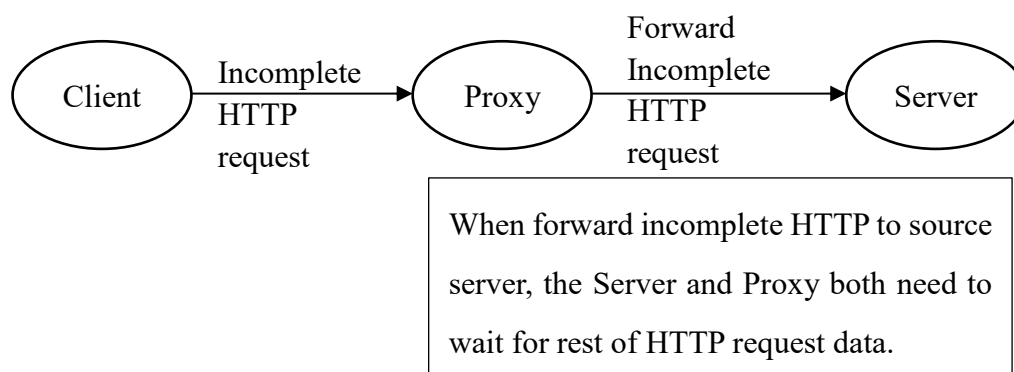


Figure 5-3 Forward incomplete HTTP request

In this paper, in the proxy cache design, will receive a complete HTTP request, and then check the cache and forward the request or direct response when cache misses. This design can greatly reduce the source server load and enhance the security of service in the event of DDoS attacks.

### 5.3.1 Overall design of proxy and cache

In order to reduce the complexity of the program, the program in the design of the appropriate increase in the proxy module and cache module coupling. When the proxy server receives a request from the client, the program first checks whether the local cache exists, according to the cache check results to determine how the next step. If the cache exists, the cache file to respond to the data, sent to the client. If the cache does not exist, the client's request is forwarded to the source server. When the source server sends a response to the proxy server, the proxy server will selectively cache the data of the source server's response to the local file and forward the response data to the source server client; the cache to the file in the data can provide the next proxy server receives the same request, directly to the client response to the cache data.

All IO operations on networks and files are asynchronous nonblocking. The overall flow chart is shown in Figure 5-4.



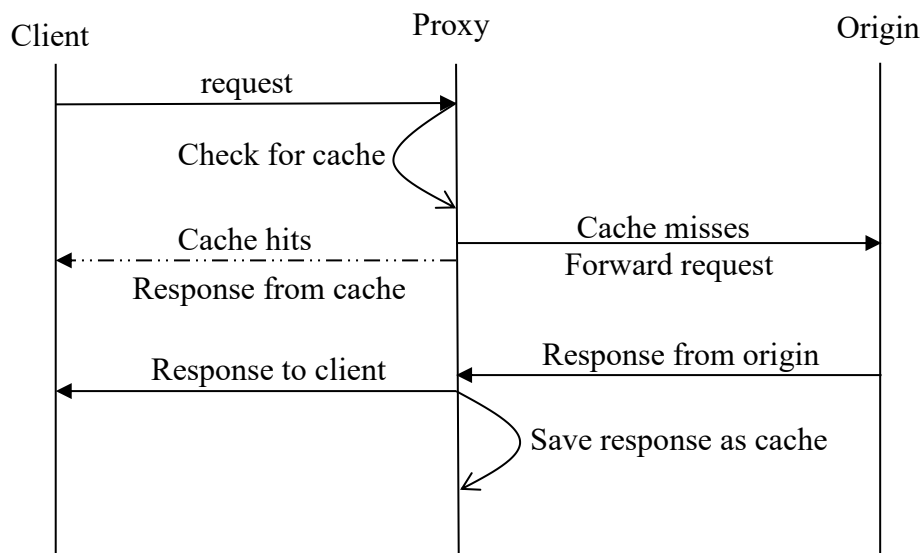


Figure 5-4 Proxy and cache flow chart

### Proxy and cache key pseudo-code

```

void read_from_client(int fd)
{
    do {
        read_request(fd);
    } while ( request not complete );    /* Read a complete HTTP request. */
    check_for_cache();                    /* Check for cache */
    if ( cache hit )                      /* Cache hit, send cache data to client. */
        send_cache();
    else                                  /* Cache miss, forward client request to source
server and wait for response. */
        forward_request();    }

void read_from_origin()
{
    forward_response();                  /* Forward response to client. */
    save_response_as_cache();    }      /* Save response as cache. */
  
```

### 5.3.2 Saving the response into cache file

In the design of saving the response data as a cache, it is necessary to write the file into a cache file as synchronous method rather than asynchronous method. If the write cache is

designed to be asynchronous IO, the order of asynchronous writes to the cache will be uncontrollable, causing the received response data to be inconsistent with the contents of the file written to the cache as shown in Figure 5-5.

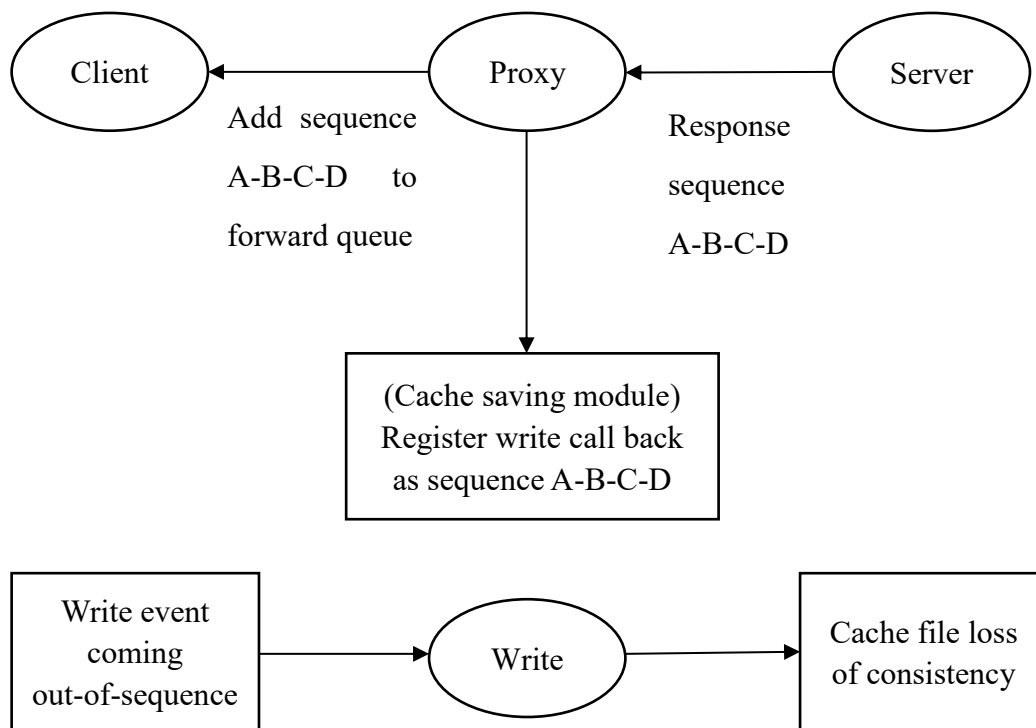


Figure 5-5 Bad design of saving cache

## 6 Implementation

The software uses a single-process mode on the implementation, and only one event loop function runs within the main function.

### 6.1 Main function implementation

The main function of the main work parsing configuration file, according to the configuration file configuration initialization of the various modules, and finally set the event\_base wait for client request. If any error occurs while parsing the configuration file, the program terminates execution and outputs an error message. If only after the implementation of the program without any error message output, the program is considered normal operation.

### 6.2 Configuration file resolution

This program currently supports only configuration log files, cache folders, comments, and source server IP addresses. For details, refer to the Appendix, Program user's manual chapter.

In order to simplify the complexity of the configuration file, the program in the configuration file format design reference to the early AT command mode, with a specific string that a configuration, to line breaks (ASCII as '\n') as the end of the flag. As mentioned earlier, the program only supports four configurations, corresponding to four strings. The configuration file starts with the string "log : "; configures the cache folder to start with the string "cache : "; configures the origin server address starts with the string "upstream : "; configure file comment start with one single pound sign (ASCII as '#').

### 6.3 Log system implementation

#### 6.3.1 Log module initialization

Log module initialization accepts the configuration file parsing module the incoming string pointer and string length of the log file configuration, the log module's initialization function opens the corresponding file according to the incoming parameters and sets the open file descriptor to non-Blocking mode, initialization file mutex.

### 6.3.2 Write the log

In order to call the log interface in other modules, the value of some variables can be written to the log, the log write function needs to support the formatted character output, the function definition needs to be defined as a parameter function. Since the log system is not a necessary module for the program, the log module supports macro-defined conditional compilation at design time to turn the log system on or off. In order to reduce the need to write to the log function of the statement block of the conditions of the compiler, so that other modules transparent write log, log write function design for the macro definition type, open or close the log module were replaced with a real log write function and empty functions.

### 6.3.3 Asynchronous mechanism

When the other log calls the log function, the log system only puts the information that needs to be logged into the heap and sets the writeback function for the log file and returns it immediately. For the caller, the call is made for a non-blocking function. The log system calls the write function at the appropriate time, writes the log information of the other modules to the file, and releases the heap space occupied by the log information.

The log system has made some processing on common exceptions, such as waiting for a write event to time out and a file mutex failed. When these two exceptions occur, the log callback function will register the log information write handle again until the log information is written successfully.

## 6.4 Proxy and cache implementation

The proxy module and the cache module are highly coupled in design, so the modules can't be implemented separately. The proxy server first reads all requests from the client, and the proxy server does the next step only if a complete request is received. The advantage of this design is that it can effectively reduce the source server by the refusal of server attacks, while ensuring that the client to the source server packet loss between the packet all transferred to the proxy server.

The proxy module receives a complete HTTP request, the entire request will be passed to the cache module, the cache module to determine whether the current request information exists in the local cache; if there is cache, the direct read the contents of the cache file sent to

the client, if it does not exist, it forwards the request to the source server, forwards the response request from the source server, and judges whether the current response is cached to the local proxy server based on the HTTP cached header.

In the implementation of the proxy and cache module, the client's request is divided into cache hit and cache miss two cases. All IO (including network IO and file IO) are all implemented as asynchronous and non-blocking.

### **6.4.1 File cache implementation**

In an HTTP request, the same host of different URLs corresponding to the information is also different. According to the URL uniqueness, the software is designed to use the HTTP request line string as the KEY, using the MD5 algorithm as the hash algorithm to compute the 32-byte MD5 value of the HTTP request line containing the URL and use that value as the response cache the file name into the program configuration cache path. MD5 as an irreversible information digest hash algorithm, has a strong anti-conflict, more suitable for large-scale Web site a large number of pages and image cache hit algorithm.

The software uses the MD5 value of the request line as the file name, writes the response data sent by the server to the configured cache path, and the next time the client accesses the same content to generate a cache hit event.

The proxy cache server, on the implementation, begins the next step when it receives a complete HTTP request, rather than the edge request. The advantage of this design is not only to reduce the TCP packet loss or HTTP request exception caused by the possibility of decline in performance of the source server, but also to simplify the complexity of proxy server design, although the response speed has slowed, but this choice is relatively speaking is desirable. When a complete client's request is received, the proxy server first calculates the MD5 value according to the request information, opens the cache file with the MD5 value as the parameter, and determines that the cache is misses if the operation fails otherwise determines that the cache is hit.

### **6.4.2 Cache hits**

Upon receipt of the client to send a request, the proxy server to determine the case of cache hit, the proxy server will read the cache file, directly to the client to return the contents of the

cache file. As with the log system, the file is written to the client socket as an asynchronous event. When the cache file is readable, the read data is placed in the client writeback cache, and Libevent will send the cached message to the client at the appropriate time. File close and other memory recovery and other clean-up work into the error event callback function, to avoid memory leaks.

### **6.4.3 Cache misses**

When the client sends a request, the proxy server needs to forward the request information of the client to the source server and forward it to the client when the source server sends the response message to the proxy server, and writes the cache file, for the next visit to the same resource can be directly returned to the cache information.

If the response request is forwarded to the client and the write cache file is designed as an unrelated asynchronous event, the CPU computing resources will be wasted when a large amount of memory and memory allocation is recovered while ensuring the consistency of the content write which will reduce software maintainability. Therefore, in the design of cache misses event when the write cache, will be forwarded to the client binding as an asynchronous event, in response to the forwarding server, while writing to the cache file.

## 7 Software test and comparison

### 7.1 Test environment

#### 7.1.1 Hardware and software environment

The host runs *nginx 1.13.0 for Windows* as the source server, the client runs *Google Chrome version 58.0.3029.110 (64-bit) for Windows* as the client, and the virtual machine of runs the software as a proxy server.

Host model: Mibook Air 12.5 inch

Hardware specifications: Intel Core m3-6y30 @ 1.51GHz, 4G DDR3 @ 1866MHz

Operating system: Microsoft Windows 10 Professional 10.0.14393

Virtualization Platform: Microsoft Hyper-V Version 10.0.14393.0

Virtual machine operating system and compilation environment: Ubuntu 16.04, Gun Make 4.1, gcc 5.4.0

#### 7.1.2 Network topology

Both system is direct connected to a router and have a static IP address as shown as Figure 7-1.

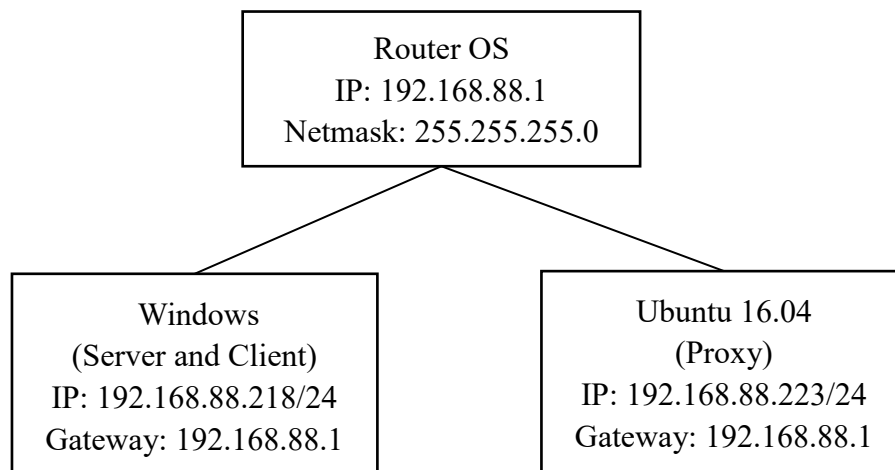


Figure 7-1 Test environment network topology

### 7.2 Test environment configuration

Source server: configure two resources, one for the html static page index.html, another for

the Hunan Institute of Engineering emblem hnie\_logo.jpg, the other keep the default.

Proxy server: The source IP address of the source server is configured as the host IP address: 192.168.88.218. The cache path is configured as: /tmp/hnie/cache/, and the log file is configured as: /tmp/hnie/ log.

## 7.3 Test method

### 7.3.1 Access the un-cached static html file

When access un-cached static html file, the client received file is illustrated in Figure 7-2.



Figure 7-2 Client access un-cached static html

The log file output is illustrated in Figure 7-3.

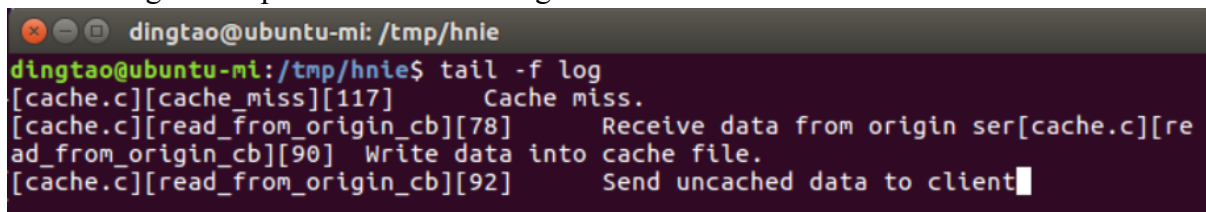


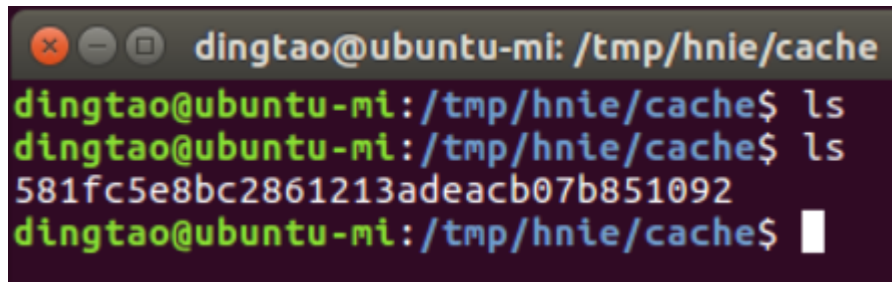
Figure 7-3 Log output when access un-cached static html

### 7.3.2 Access the cached static html file

After the previous access to the index.html static page, the proxy server has successfully cached the source server's response results locally, caching the file as shown in Figure 7-4. Once

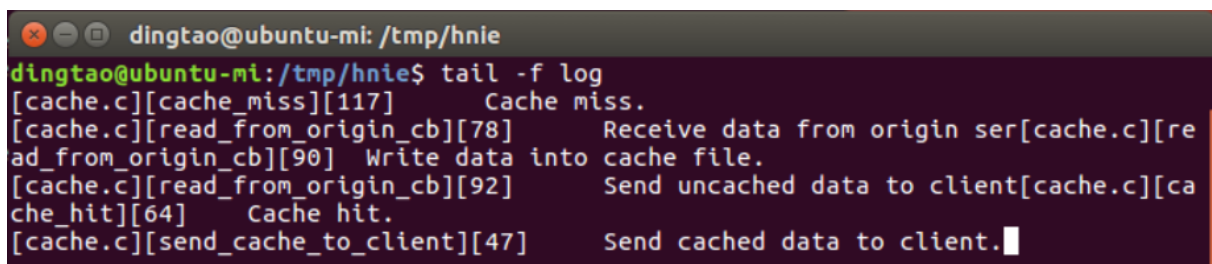


again access index.html, get the results as the same as Figure 7-2. At this point, the log file output cache hit (as shown in Figure 7-5), to prove the program works just fine.



```
dingtao@ubuntu-mi: /tmp/hnie/cache
dingtao@ubuntu-mi:/tmp/hnie/cache$ ls
dingtao@ubuntu-mi:/tmp/hnie/cache$ ls
581fc5e8bc2861213adeacb07b851092
dingtao@ubuntu-mi:/tmp/hnie/cache$
```

Figure 7-4 Index.html cached in local file



```
dingtao@ubuntu-mi: /tmp/hnie
dingtao@ubuntu-mi:/tmp/hnie$ tail -f log
[cache.c][cache_miss][117]      Cache miss.
[cache.c][read_from_origin_cb][78]  Receive data from origin ser[cache.c][re
ad_from_origin_cb][90]  Write data into cache file.
[cache.c][read_from_origin_cb][92]  Send uncached data to client[cache.c][ca
che_hit][64]      Cache hit.
[cache.c][send_cache_to_client][47]  Send cached data to client.
```

Figure 7-5 Log output when access cached static html

### 7.3.3 Access the un-cached picture

Accessing un-cached pictures is basically no different from accessing static html pages that are not cached. To test program correctness, the image cache needs to be tested after testing a static html page. Client access picture results shown in Figure 7-6, proxy server log output as shown in Figure 7-7.



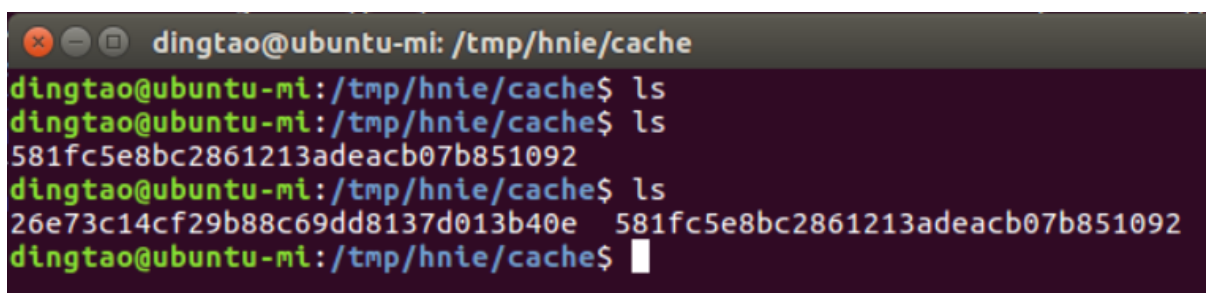
Figure 7-6 Client access un-cached picture

```
dingtao@ubuntu-mi: /tmp/hnie
dingtao@ubuntu-mi:/tmp/hnie$ tail -f log
[cache.c][cache_miss][117]      Cache miss.
[cache.c][read_from_origin_cb][78]    Receive data from origin ser[cache.c][re
ad_from_origin_cb][90]  Write data into cache file.
[cache.c][read_from_origin_cb][92]    Send uncached data to client[cache.c][ca
che_hit][64]      Cache hit.
[cache.c][send_cache_to_client][47]    Send cached data to client.[cache.c][cac
he_miss][117]      Cache miss.
[cache.c][read_from_origin_cb][78]    Receive data from origin ser[cache.c][re
ad_from_origin_cb][90]  Write data into cache file.
[cache.c][read_from_origin_cb][92]    Send uncached data to client
```

Figure 7-7 Log output when access un-cached picture

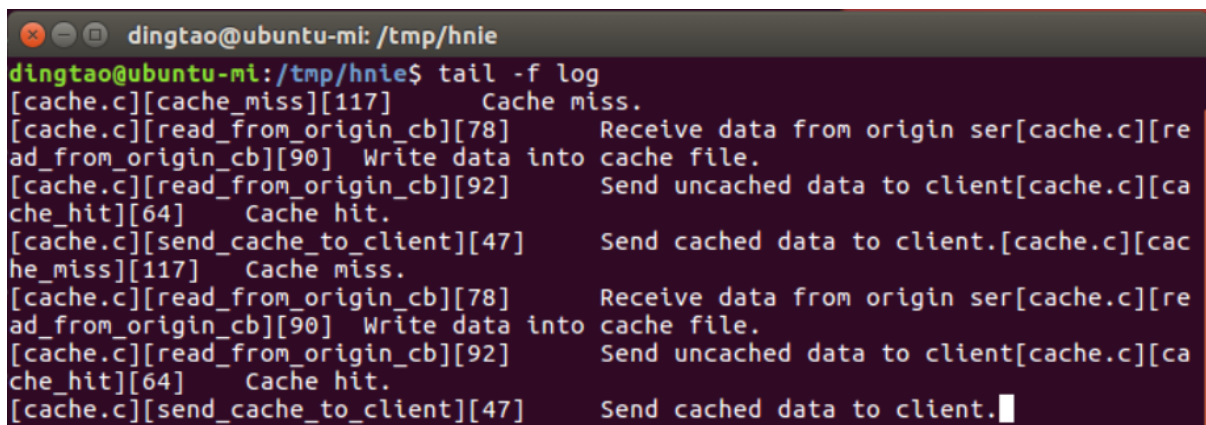
### 7.3.4 Access the cached picture

After accessing the un-cached image, the proxy server has cached the image file locally (shown in Figure 7-8). When access the image again, and the client gets the same result (shown in Figure 7-6). Proxy server log output as shown in Figure 7-9, to prove the program can work when proxy the picture file.



```
dingtao@ubuntu-mi: /tmp/hnie/cache
dingtao@ubuntu-mi:/tmp/hnie/cache$ ls
dingtao@ubuntu-mi:/tmp/hnie/cache$ ls
581fc5e8bc2861213adeacb07b851092
dingtao@ubuntu-mi:/tmp/hnie/cache$ ls
26e73c14cf29b88c69dd8137d013b40e 581fc5e8bc2861213adeacb07b851092
dingtao@ubuntu-mi:/tmp/hnie/cache$
```

Figure 7-8 Hnie\_logo.jpg cached in local file



```
dingtao@ubuntu-mi: /tmp/hnie
dingtao@ubuntu-mi:/tmp/hnie$ tail -f log
[cache.c][cache_miss][117] Cache miss.
[cache.c][read_from_origin_cb][78] Receive data from origin ser[cache.c][re
ad_from_origin_cb][90] Write data into cache file.
[cache.c][read_from_origin_cb][92] Send uncached data to client[cache.c][ca
che_hit][64] Cache hit.
[cache.c][send_cache_to_client][47] Send cached data to client.[cache.c][cac
he_miss][117] Cache miss.
[cache.c][read_from_origin_cb][78] Receive data from origin ser[cache.c][re
ad_from_origin_cb][90] Write data into cache file.
[cache.c][read_from_origin_cb][92] Send uncached data to client[cache.c][ca
che_hit][64] Cache hit.
[cache.c][send_cache_to_client][47] Send cached data to client.
```

Figure 7-9 Log output when access cached picture

## 7.4 Comparison

In order to test the performance of this software, this paper uses nginx as the comparison object, and simulates the high concurrent environment to test the software and nginx respectively.

### 7.4.1 Comparison method

Run *nginx 1.13.0* and this software in Ubuntu 16.04 system virtual machine with 1 core 512MB memory, and use 100, 500, 1000, 2000, 4000, 6000, 8000, 10000 concurrent numbers to simulate user request data to initiate HTTP request to server, record the number of success response.

### 7.4.2 Comparison result

All the test source code and raw data can be find in Evaluation data chapter, the result shown as Figure 7-10.

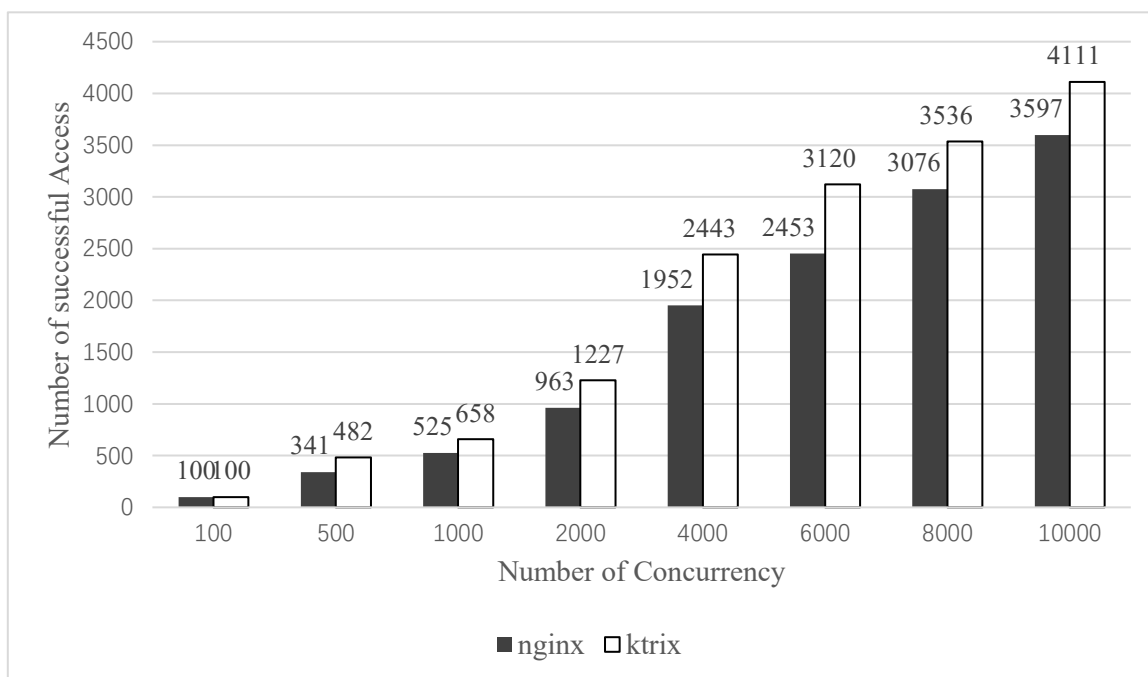


Figure 7-10 High concurrent comparison result

The test results show that this software in a high concurrent environment, the request success rate is slightly higher than the nginx. The reason is that nginx over resolution HTTP packets, taking up too much CPU computing resources lead to performance degradation.

## 8 Summary

The general proxy server, used only for proxy connections to the Internet over the Internet, the client must specify the proxy server and send the HTTP request that was originally sent directly to the Web server to the proxy server. Since the hosts on the external network are not configured and use this proxy server, the generic proxy server is also designed to search for multiple indeterminate servers on the Internet instead of accessing a fixed request for multiple clients on the Internet server, so the normal Web proxy server does not support external access requests to the internal network. When a proxy server can proxy the host on the external network, access to the internal network, this proxy service is called reverse proxy service. At this point the proxy server external performance for a Web server, the external network can simply treat it as a standard Web server without the need for a specific configuration. The difference is that the server does not save any web page real data, all static web pages or CGI programs are stored on the internal Web server. So, the attack on the reverse proxy server does not make the page information is damaged, thus enhancing the Web server security.

There is no conflict between the reverse proxy mode and the packet filtering mode or the normal proxy mode. Therefore, users can use both methods in the firewall device, where the reverse proxy is used when the external network accesses the internal network, forward proxy, or packet filtering used to deny other external access methods and provide internal network access to external networks. So, users can combine these methods to provide the best way to secure access.

Using a reverse proxy server with caching capabilities can reduce the load on the original Web server. The reverse proxy server assumes a static page request for the original Web server to prevent the original server from overloading. It is located between the local Web server and the Internet, handle all requests to the Web server, to prevent the Web server and the Internet direct communication. If the page requested by the Internet user is buffered on the proxy server, the proxy server sends the buffered content directly to the user. If there is no buffer, the first request to the Web server, retrieve the data, the local cache and then sent to the user. This way by reducing the number of requests to the Web server to reduce the Web server load.

## Bibliography

- [1]. Xie Xiren. *Computer Network[M]*. Sixth Edition. Beijing, Electronic Industry Press. June 2013.
- [2]. Gourley, D. *et al.* *HTTP: The Definitive Guide[M]*. Sebastopol, O'Reilly Media, Inc. September 2002.
- [3]. T. Berners-Lee, R. Fielding and H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*, IETF RFC 1945, May 1996; [www.rfc-editor.org/rfc/rfc1945.txt](http://www.rfc-editor.org/rfc/rfc1945.txt).
- [4]. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, *Hypertext Transfer Protocol - HTTP/1.1*, IETF RFC 2068, January 1997; [www.rfc-editor.org/rfc/rfc2068.txt](http://www.rfc-editor.org/rfc/rfc2068.txt).
- [5]. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, IETF RFC 2616, June 1999; [www.rfc-editor.org/rfc/rfc2616.txt](http://www.rfc-editor.org/rfc/rfc2616.txt).
- [6]. R. Fielding, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, IETF RFC 7230, June 2014; [www.rfc-editor.org/rfc/rfc7230.txt](http://www.rfc-editor.org/rfc/rfc7230.txt).
- [7]. R. Fielding, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, IETF RFC 7231, June 2014; [www.rfc-editor.org/rfc/rfc7231.txt](http://www.rfc-editor.org/rfc/rfc7231.txt).
- [8]. R. Fielding, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*, IETF RFC 7232, June 2014; [www.rfc-editor.org/rfc/rfc7232.txt](http://www.rfc-editor.org/rfc/rfc7232.txt).
- [9]. R. Fielding, Ed., Y. Lafon, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*, IETF RFC 7233, June 2014; [www.rfc-editor.org/rfc/rfc7233.txt](http://www.rfc-editor.org/rfc/rfc7233.txt).
- [10]. R. Fielding, Ed., M. Nottingham, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Caching*, IETF RFC 7234, June 2014; [www.rfc-editor.org/rfc/rfc7234.txt](http://www.rfc-editor.org/rfc/rfc7234.txt).
- [11]. R. Fielding, Ed. and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Authentication*, IETF RFC 7235, June 2014; [www.rfc-editor.org/rfc/rfc7235.txt](http://www.rfc-editor.org/rfc/rfc7235.txt).
- [12]. M. Belshe, R. Peon and M. Thomson, Ed., *Hypertext Transfer Protocol Version 2 (HTTP/2)*, IETF RFC 7240, May 2015; [www.rfc-editor.org/rfc/rfc7240.txt](http://www.rfc-editor.org/rfc/rfc7240.txt).
- [13]. R. Peon and H. Ruellan, *HPACK: Header Compression for HTTP/2*, IETF RFC 7241, May 2015; [www.rfc-editor.org/rfc/rfc7241.txt](http://www.rfc-editor.org/rfc/rfc7241.txt).
- [14]. W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols[M]*. New Jersey. Addison-Wesley. 1994.
- [15]. W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX*

- Environment[M]*. Third Edition. New Jersey. Pearson Education, Inc. May 2013.
- [16]. W. Richard Stevens, Bill Fenner and Andrew M. Rudoff. *UNIX Network Programming Volume 1: The Sockets Networking API[M]*. Third Edition. New Jersey. Pearson Education, Inc. November 2003.
- [17]. Nick Mathewson, *Fast portable non-blocking network programming with Libevent*, Libevent.org, Jan 2012; [www.wangafu.net/~nickm/libevent-book/TOC.html](http://www.wangafu.net/~nickm/libevent-book/TOC.html).

## **Acknowledgements**

I am greatly indebted to my supervisor, Professor Peng Meng, for his valuable instructions and suggestions on my thesis as well as his careful reading of the manuscript. I feel grateful to all the teachers in the Computer and Communication Department of Hunan Institute of Engineering who once offered me valuable courses and advice during my study. My sincere thanks are also given to my colleagues in Shenzhen, Chen Yi and Guo Xiong, from whose lectures I benefited greatly.

Last but not least, I owe much to my friends and classmates for their valuable suggestions and critiques which are of help and importance in making the thesis a reality.



## Appendix

### Program user's manual

There is no option of this program, just one parameter of config file that needed.

`./thesis thesis.conf`

### Config

This program current support four feature, comment, log file, cache directory and original web server's IP address.

### Comment

Type one or more “#” in the start of one line, just like shell script.

### Log

Usage: `log : log_file`

Example: `log : /tmp/hnie/log`

Note: There are two space between “:” can NOT be ignored.

### Cache

Usage: `cache : cache_dir`

Example: `cache : /tmp/hnie/cache/`

Note: There are two space between “:” can NOT be ignored.

### Original web server's address

Usage: `upstream : IPv4 address`

Example: `upstream : 192.168.0.1`

Note: There are two space between “:” can NOT be ignored. Current support only ipv4 address, may support ipv6 address and domain name.

## Source Code

main.c

```
#include <stdio.h>
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>

#include <event2/event.h>

#include "log.h"
#include "http.h"
#include "cache.h"
#include "proxy.h"
#include "config.h"
#include "global.h"

#define GLOBAL_BUFFER_SIZE 1024

char g_buffer[GLOBAL_BUFFER_SIZE] = {'\0'};
struct event_base * g_base = NULL;
struct evconnlistener * g_listener = NULL;
struct sockaddr_in g_local_addr, g_remote_addr;

void init(void)
{
    struct event_config *cfg = event_config_new();
    if ( !cfg )
    {
        perror("event_config_new() Error!\n");
        exit(-1);
    }
    /** Use avoid method to avoid epoll function.
     * Due to unsupported local file descriptor on epoll.
     */
    event_config_avoid_method(cfg, "epoll");
    g_base = event_base_new_with_config(cfg);
    if ( !g_base )
    {
        perror("event_base_new failed!\n");
        exit(-1);
    }
}
```

```
    return ;
}

int main(int argc, char *argv[])
{
    if ( 2 != argc )
    {
        perror("usage : ktrix file.conf\n");
        exit(-1);
    }
    init();
    config_init(argv[1]);
    event_base_dispatch(g_base);
    return 0;
}
```

## log.c

```
/**
 * This file is published under no specific lisenice, you can use this
file
 * in any usage now.
 *
 * Copyright (C) 2017 Ding Tao, Hunan Institute of Engineering All right
reserved.
 *
 * */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#include <unistd.h>
#include <fcntl.h>

#include <pthread.h>

#include <event2/event.h>

#include "log.h"
#include "global.h"

#define LOG_DEFAULT_BUFF_SIZE 64
```

```
static const struct timeval g_timeout = {5, 0};

int g_log_fd = 0;
static pthread_mutex_t g_log_mutex;

void log_cb(int fd, short what, void * arg)
{
    if ( !arg )
    {
        perror("In func log_cb, arg == NULL!\n");
        exit(-1);
    }
    if ( what & EV_WRITE )
    {
        if ( 0 != pthread_mutex_trylock(&g_log_mutex) )
        {
            event_base_once(g_base, g_log_fd, EV_WRITE, log_cb, arg,
&g_timeout);
            return ;
        }
        /* At this point, mutex_trylock is successfully locked. */
        char *str = (char *)arg;
        ssize_t left = strlen(str), n = 0;
        do
        {
            n = write(g_log_fd, str, left);
            if ( n > 0 )
            {
                left -= n;
            }
        }while (left);
        /* Unlock the log file. */
        pthread_mutex_unlock(&g_log_mutex);
        free(str);
        return ;
    }
    if ( what & EV_TIMEOUT )
    {
        /* If this event didn't trigger when timeout, reset current event.
*/
        event_base_once(g_base, g_log_fd, EV_WRITE, log_cb, arg,
&g_timeout);
        return ;
    }
}
```

```
void log_init(char * buff, int len)
{
    char path[128] = {0};
    if ( !buff )
    {
        printf("No log file!\n");
        exit(-1);
    }
    for ( int i = 0; i < len; ++i )
    {
        path[i] = buff[i];
    }
    g_log_fd = open(path, O_WRONLY | O_APPEND);
    if ( g_log_fd <= 0 )
    {
        printf("Can not open file %s!\n", path);
        exit(-1);
    }
    fcntl(g_log_fd, F_SETFL, fcntl(g_log_fd, F_GETFL) | O_NONBLOCK);

    if ( 0 != pthread_mutex_init(&g_log_mutex, NULL) )
    {
        printf("g_log_rwlock initial error!\n");
        exit(-1);
    }
    return ;
}

void _log(log_level_t level, const char * file, const char * func,
uint32_t line, const char * restrict format, ...)
{
    char *str = NULL;
    int len;
    va_list ap;

    str = malloc(LOG_DEFAULT_BUFF_SIZE);
    if ( !str )
    {
        printf("Out of memory!\n");
        exit(-1);
    }
    len = sprintf(str, "[%s][%s][%d]\t", file, func, line);

    va_start(ap, format);
```

```
    vsnprintf( str + len, LOG_DEFAULT_BUFF_SIZE - len, format, ap );
    va_end(ap);

    event_base_once(g_base, g_log_fd, EV_WRITE, log_cb, (void*)str,
&g_timeout);
    return ;
}

void _log_invalid(log_level_t level, const char * file, const char *
func, uint32_t line, const char * restrict format, ...)
{
    return ;
}
```

### config.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "config.h"
#include "log.h"
#include "cache.h"
#include "http.h"
#include "global.h"

#define CONFIG_BUFF_SIZE 128

void config_init(const char * file)
{
    int config_file_fd, n, total, pos, start;
    char buff[CONFIG_BUFF_SIZE];
    if ( !file )
    {
        perror("Can NOT find the config file!\n");
        exit(-1);
    }
    config_file_fd = open(file, O_RDONLY);
    if ( config_file_fd <= 0 )
```

```

{
    perror("Open config file error!\n");
    exit(-1);
}

total = 0;
pos = 0;
while ( ( n = read(config_file_fd, &buff[total], CONFIG_BUFF_SIZE-
1) ) > 0 )
{
    total += n;
    buff[n] = '\0';
    for ( ; pos < total && buff[pos] != '\0'; ++pos )
    {
        if ( buff[pos] == '#' ) /* Comment in one line. */
        {
            /* Skip current line. */
            while ( pos < total && buff[pos] != '\0' )
            {
                if ( buff[pos] == '\n' )
                {
                    break;
                }
                ++pos;
            }
        }
        if ( strncmp(&buff[pos], "log : ", 6) == 0 )
        {
            pos += 6;
            start = pos;
            while ( pos < total && buff[pos] != '\0' && buff[pos] !=
'\n' )
            {
                ++pos;
            }
            log_init(&buff[start], pos-start);
        }
        else if ( strncmp(&buff[pos], "cache : ", 8) == 0 )
        {
            pos += 8;
            start = pos;
            while ( pos < total && buff[pos] != '\0' && buff[pos] !=
'\n' )
            {
                ++pos;
            }
        }
    }
}

```

```

    }
    cache_init(&buff[start], pos-start);
}
else if ( strcmp(&buff[pos], "upstream : ", 11) == 0 )
{
    pos += 11;
    start = pos;
    while ( pos < total && buff[pos] != '\0' && buff[pos] !=
'\n' )
    {
        ++pos;
    }
    http_init(&buff[start], pos-start);
}
}/* End of for. */
}/* End of while. */
return ;
}

```

## http.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "proxy.h"
#include "http.h"
#include "log.h"
#include "global.h"

void http_init(char * buff, int len)
{
    char addr[16] = {'\0'};
    for ( int i = 0; i < len; ++i )
    {
        addr[i] = buff[i];
    }
    if ( inet_pton(AF_INET, (const char*)addr,
&g_remote_addr.sin_addr) != 1 )
    {
        perror("Wrong ip address!\n");
    }
}

```



```
        exit(-1);
    }

    memset(&g_local_addr, 0, sizeof(struct sockaddr_in));
    g_local_addr.sin_family = AF_INET;
    g_local_addr.sin_addr.s_addr = htonl(0);
    g_local_addr.sin_port = htons(8080);

    //memset(&g_remote_addr, 0, sizeof(struct sockaddr_in));
    g_remote_addr.sin_family = AF_INET;
    g_remote_addr.sin_port = htons(80);

    proxy_init();
    return ;
}
```

### cache.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <unistd.h>
#include <fcntl.h>

#include <event2/event.h>
#include <event2/buffer.h>
#include <event2/bufferevent.h>

#include "md5.h"
#include "http.h"
#include "cache.h"
#include "log.h"
#include "global.h"

char g_cache_dir[128] = {0};

void cache_init(char * buff, int len)
{
    for ( int i = 0; i < len; ++i )
    {
        g_cache_dir[i] = buff[i];
    }
    return ;
}
```

```
/* Watch out!! This function never been tested before, be careful when
call it. */
void get_request_line_md5(unsigned char *encrypt, char *hexstr)
{
    MD5_CTX md5;
    uint8_t decrypt[16];

    MD5Init(&md5);
    MD5Update(&md5, encrypt, strlen((char *)encrypt));
    MD5Final(&md5, decrypt);
    for ( int i = 0; i < 16; ++i )
    {
        sprintf(&hexstr[i*2], "%02x", decrypt[i]);
    }
    return ;
}

void send_cache_to_client(int fd, short events, void * ptr)
{
    LOG(LOG_LEVEL_INFO, "Send cached data to client.\n");
    if ( events & EV_READ )
    {
        struct evbuffer * buff = bufferevent_get_output((struct
bufferevent *)ptr);
        int n = 0;
        char str[1024] = {0};
        while ( ( n = read(fd, str, 1024) ) > 0 )
        {
            evbuffer_add(buff, str, n);
        }
        close(fd);
    }
    /* Ignore all other events. */
}

void cache_hit(struct bufferevent * bev, int fd)
{
    LOG(LOG_LEVEL_INFO, "Cache hit.\n");
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
    event_base_once(g_base, fd, EV_READ, send_cache_to_client, bev,
NULL);
    return ;
}
```

```
typedef struct
{
    struct bufferevent * bev;
    int fd;
}origin_arg_t;

void read_from_origin_cb(struct bufferevent * bev, void * ptr)
{
    LOG(LOG_LEVEL_INFO, "Receive data from origin server.\n");
    origin_arg_t * arg = ptr;
    if ( !arg )
    {
        LOG(LOG_LEVEL_FATAL, "No arg!\n");
        exit(-1);
    }
    struct evbuffer * origin = bufferevent_get_input(bev);
    struct evbuffer * client = bufferevent_get_output(arg->bev);
    size_t str_len = evbuffer_get_length(origin);
    char * str = malloc(str_len);
    evbuffer_copyout(origin, str, str_len);
    LOG(LOG_LEVEL_INFO, "Write data into cache file.\n");
    evbuffer_add_buffer(client, origin);
    LOG(LOG_LEVEL_INFO, "Send uncached data to client.\n");
    for ( int i = 0, n = 0; i < str_len; i += n )
    {
        n = write(arg->fd, str, str_len);
    }
    free(str);
}

void event_of_origin_cb(struct bufferevent * bev, short events, void *
ptr)
{
    if ( events & ( BEV_EVENT_EOF | BEV_EVENT_ERROR) )
    {
        if ( events | BEV_EVENT_EOF )
        {
            bufferevent_free(bev);
        }
        origin_arg_t * arg = ptr;
        close(arg->fd);
        free(arg);
    }
}
```

```

void cache_miss(struct bufferevent * bev, int fd)
{
    LOG(LOG_LEVEL_INFO, "Cache miss.\n");
    struct bufferevent * bev_out = bufferevent_socket_new(g_base, -1,
BEV_OPT_CLOSE_ON_FREE | BEV_OPT_DEFER_CALLBACKS);
    if ( !bev_out )
    {
        LOG(LOG_LEVEL_FATAL, "Create bufferevent failed.");
        bufferevent_free(bev_out);
        exit(-1);
    }
    if ( bufferevent_socket_connect(bev_out, (struct sockaddr
*)&g_remote_addr, sizeof(struct sockaddr_in)) < 0 )
    {
        LOG(LOG_LEVEL_FATAL, "Can NOT connect to remote server.");
        exit(-1);
    }

    /* At this point, we are managed to receive data from origin. */

    origin_arg_t * arg = malloc(sizeof(origin_arg_t));
    if ( !arg )
    {
        LOG(LOG_LEVEL_FATAL, "No memory.");
        exit(-1);
    }
    arg->bev = bev;
    arg->fd = fd;

    bufferevent_setcb(bev_out, read_from_origin_cb, NULL,
event_of_origin_cb, arg);
    bufferevent_enable(bev_out, EV_READ | EV_WRITE);

    struct evbuffer * src = bufferevent_get_input(bev);
    struct evbuffer * dst = bufferevent_get_output(bev_out);
    evbuffer_add_buffer(dst, src);
}

```

md5.c

```

#include <memory.h>
#include "md5.h"

unsigned char PADDING[] =
{

```

```

0x80,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};

void MD5Init(MD5_CTX *context)
{
    context->count[0] = 0;
    context->count[1] = 0;
    context->state[0] = 0x67452301;
    context->state[1] = 0xEFCDAB89;
    context->state[2] = 0x98BADCFE;
    context->state[3] = 0x10325476;
}

void MD5Update(MD5_CTX *context,unsigned char *input,unsigned int
inputlen)
{
    unsigned int i = 0,index = 0,partlen = 0;
    index = (context->count[0] >> 3) & 0x3F;
    partlen = 64 - index;
    context->count[0] += inputlen << 3;
    if ( context->count[0] < (inputlen << 3) )
    {
        context->count[1]++;
    }
    context->count[1] += inputlen >> 29;

    if ( inputlen >= partlen )
    {
        memcpy(&context->buffer[index],input,partlen);
        MD5Transform(context->state,context->buffer);
        for(i = partlen;i+64 <= inputlen;i+=64)
        {
            MD5Transform(context->state,&input[i]);
        }
        index = 0;
    }
    else
    {
        i = 0;
    }
    memcpy(&context->buffer[index],&input[i],inputlen-i);
}

void MD5Final(MD5_CTX *context,unsigned char digest[16])
{

```

```

    unsigned int index = 0, padlen = 0;
    unsigned char bits[8];
    index = (context->count[0] >> 3) & 0x3F;
    padlen = (index < 56) ? (56-index) : (120-index);
    MD5Encode(bits, context->count, 8);
    MD5Update(context, PADDING, padlen);
    MD5Update(context, bits, 8);
    MD5Encode(digest, context->state, 16);
}

void MD5Encode(unsigned char *output, unsigned int *input, unsigned int
len)
{
    unsigned int i = 0, j = 0;
    while ( j < len )
    {
        output[j] = input[i] & 0xFF;
        output[j+1] = (input[i] >> 8) & 0xFF;
        output[j+2] = (input[i] >> 16) & 0xFF;
        output[j+3] = (input[i] >> 24) & 0xFF;
        i++;
        j+=4;
    }
}

void MD5Decode(unsigned int *output, unsigned char *input, unsigned int
len)
{
    unsigned int i = 0, j = 0;
    while ( j < len )
    {
        output[i] = (input[j]) |
                    (input[j+1] << 8) |
                    (input[j+2] << 16) |
                    (input[j+3] << 24);
        i++;
        j+=4;
    }
}

void MD5Transform(unsigned int state[4], unsigned char block[64])
{
    unsigned int a = state[0];
    unsigned int b = state[1];
    unsigned int c = state[2];
    unsigned int d = state[3];
    unsigned int x[64];
    MD5Decode(x, block, 64);

```

```
FF(a, b, c, d, x[ 0], 7, 0xd76aa478); /* 1 */
FF(d, a, b, c, x[ 1], 12, 0xe8c7b756); /* 2 */
FF(c, d, a, b, x[ 2], 17, 0x242070db); /* 3 */
FF(b, c, d, a, x[ 3], 22, 0xc1bdceee); /* 4 */
FF(a, b, c, d, x[ 4], 7, 0xf57c0faf); /* 5 */
FF(d, a, b, c, x[ 5], 12, 0x4787c62a); /* 6 */
FF(c, d, a, b, x[ 6], 17, 0xa8304613); /* 7 */
FF(b, c, d, a, x[ 7], 22, 0xfd469501); /* 8 */
FF(a, b, c, d, x[ 8], 7, 0x698098d8); /* 9 */
FF(d, a, b, c, x[ 9], 12, 0x8b44f7af); /* 10 */
FF(c, d, a, b, x[10], 17, 0xffff5bb1); /* 11 */
FF(b, c, d, a, x[11], 22, 0x895cd7be); /* 12 */
FF(a, b, c, d, x[12], 7, 0x6b901122); /* 13 */
FF(d, a, b, c, x[13], 12, 0xfd987193); /* 14 */
FF(c, d, a, b, x[14], 17, 0xa679438e); /* 15 */
FF(b, c, d, a, x[15], 22, 0x49b40821); /* 16 */

/* Round 2 */
GG(a, b, c, d, x[ 1], 5, 0xf61e2562); /* 17 */
GG(d, a, b, c, x[ 6], 9, 0xc040b340); /* 18 */
GG(c, d, a, b, x[11], 14, 0x265e5a51); /* 19 */
GG(b, c, d, a, x[ 0], 20, 0xe9b6c7aa); /* 20 */
GG(a, b, c, d, x[ 5], 5, 0xd62f105d); /* 21 */
GG(d, a, b, c, x[10], 9, 0x2441453); /* 22 */
GG(c, d, a, b, x[15], 14, 0xd8a1e681); /* 23 */
GG(b, c, d, a, x[ 4], 20, 0xe7d3fbc8); /* 24 */
GG(a, b, c, d, x[ 9], 5, 0x21e1cde6); /* 25 */
GG(d, a, b, c, x[14], 9, 0xc33707d6); /* 26 */
GG(c, d, a, b, x[ 3], 14, 0xf4d50d87); /* 27 */
GG(b, c, d, a, x[ 8], 20, 0x455a14ed); /* 28 */
GG(a, b, c, d, x[13], 5, 0xa9e3e905); /* 29 */
GG(d, a, b, c, x[ 2], 9, 0xfcefa3f8); /* 30 */
GG(c, d, a, b, x[ 7], 14, 0x676f02d9); /* 31 */
GG(b, c, d, a, x[12], 20, 0x8d2a4c8a); /* 32 */

/* Round 3 */
HH(a, b, c, d, x[ 5], 4, 0xffffa3942); /* 33 */
HH(d, a, b, c, x[ 8], 11, 0x8771f681); /* 34 */
HH(c, d, a, b, x[11], 16, 0x6d9d6122); /* 35 */
HH(b, c, d, a, x[14], 23, 0xfde5380c); /* 36 */
HH(a, b, c, d, x[ 1], 4, 0xa4beea44); /* 37 */
HH(d, a, b, c, x[ 4], 11, 0x4bdecfa9); /* 38 */
HH(c, d, a, b, x[ 7], 16, 0xf6bb4b60); /* 39 */
HH(b, c, d, a, x[10], 23, 0xbebfb70); /* 40 */
HH(a, b, c, d, x[13], 4, 0x289b7ec6); /* 41 */
```

```
HH(d, a, b, c, x[ 0], 11, 0xeeaa127fa); /* 42 */
HH(c, d, a, b, x[ 3], 16, 0xd4ef3085); /* 43 */
HH(b, c, d, a, x[ 6], 23, 0x4881d05); /* 44 */
HH(a, b, c, d, x[ 9], 4, 0xd9d4d039); /* 45 */
HH(d, a, b, c, x[12], 11, 0xe6db99e5); /* 46 */
HH(c, d, a, b, x[15], 16, 0x1fa27cf8); /* 47 */
HH(b, c, d, a, x[ 2], 23, 0xc4ac5665); /* 48 */

/* Round 4 */
II(a, b, c, d, x[ 0], 6, 0xf4292244); /* 49 */
II(d, a, b, c, x[ 7], 10, 0x432aff97); /* 50 */
II(c, d, a, b, x[14], 15, 0xab9423a7); /* 51 */
II(b, c, d, a, x[ 5], 21, 0xfc93a039); /* 52 */
II(a, b, c, d, x[12], 6, 0x655b59c3); /* 53 */
II(d, a, b, c, x[ 3], 10, 0x8f0ccc92); /* 54 */
II(c, d, a, b, x[10], 15, 0xffeff47d); /* 55 */
II(b, c, d, a, x[ 1], 21, 0x85845dd1); /* 56 */
II(a, b, c, d, x[ 8], 6, 0x6fa87e4f); /* 57 */
II(d, a, b, c, x[15], 10, 0xfe2ce6e0); /* 58 */
II(c, d, a, b, x[ 6], 15, 0xa3014314); /* 59 */
II(b, c, d, a, x[13], 21, 0x4e0811a1); /* 60 */
II(a, b, c, d, x[ 4], 6, 0xf7537e82); /* 61 */
II(d, a, b, c, x[11], 10, 0xbd3af235); /* 62 */
II(c, d, a, b, x[ 2], 15, 0x2ad7d2bb); /* 63 */
II(b, c, d, a, x[ 9], 21, 0xeb86d391); /* 64 */
state[0] += a;
state[1] += b;
state[2] += c;
state[3] += d;
}
```

proxy.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>

#include <sys/socket.h>
#include <netinet/in.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```



```
#include <event2/event.h>
#include <event2/buffer.h>
#include <event2/bufferevent.h>
#include <event2/listener.h>
#include <event2/util.h>

#include "log.h"
#include "proxy.h"
#include "cache.h"
#include "global.h"

void read_client_cb(struct bufferevent *, void *);
void event_client_cb(struct bufferevent *, short, void *);

void cache_check(struct bufferevent * bev, char * str)
{
    struct evbuffer * evbuff = bufferevent_get_input(bev);
    size_t request_line_len;
    char * request_line = evbuffer_readln(evbuff, &request_line_len,
    EVBUFFER_EOL_CRLF_STRICT);

    /* When use evbuffer_readln, the first line will be remove from
    evbuffer, so we just simply add it back. */
    evbuffer_prepend(evbuff, "\r\n", 2);
    evbuffer_prepend(evbuff, request_line, request_line_len);

    if ( !request_line )
    {
        LOG(LOG_LEVEL_FATAL, "CRLF error!");
        exit(-1);
    }
    get_request_line_md5((unsigned char *)request_line, str);
    free(request_line);
}

void read_client_cb(struct bufferevent * bev, void * ptr)
{
    if ( !bev )
    {
        LOG(LOG_LEVEL_FATAL, "bev == NULL!");
        exit(-1);
    }

    struct evbuffer * evbuff = bufferevent_get_input(bev);
```

```

    struct evbuffer_ptr pos = evbuffer_search(evbuff, "\r\n\r\n", 4,
NULL);
    if ( pos.pos == -1 )
    {
        /* Not finished yet. */
        return ;
    }

    char str[128] = {0};
    strcpy(str, g_cache_dir);
    int len = strlen(str);
    int fd = 0;
    cache_check(bev, &str[len]);
    if ( ( fd = open(str, O_RDONLY) ) > 0 )
    {
        cache_hit(bev, fd);
    }
    else
    {
        /* Can not open file, create new file. */
        fd = open(str, O_WRONLY | O_CREAT | O_NONBLOCK, S_IRUSR | S_IWUSR
| S_IRGRP | S_IWGRP | S_IROTH);
        if ( fd <= 0 )
        {
            LOG(LOG_LEVEL_FATAL, "Create cache file error.");
            exit(-1);
        }
        cache_miss(bev, fd);
    }
}

void event_client_cb(struct bufferevent * bev, short events, void * ptr)
{
    if ( !ptr )
    {
        LOG(LOG_LEVEL_FATAL, "ptr == NULL!");
        exit(-1);
    }
    if ( events | ( BEV_EVENT_EOF | BEV_EVENT_ERROR ) )
    {
        if ( events | BEV_EVENT_EOF )
        {
            LOG(LOG_LEVEL_INFO, "%s called.\n", __func__);
            //bufferevent_free(bev);
            //bufferevent_enable(bev, 0);

```

```
    }
    /* No implamation here. */
}
return ;
}

void accept_cb(struct evconnlistener * listener, evutil_socket_t sock,
struct sockaddr * addr, int len, void * ptr)
{
    struct bufferevent * bev_in = NULL;

    bev_in = bufferevent_socket_new(g_base, sock, BEV_OPT_CLOSE_ON_FREE
| BEV_OPT_DEFER_CALLBACKS);
    if ( !bev_in )
    {
        LOG(LOG_LEVEL_FATAL, "Create bufferevent failed.");
        exit(-1);
    }

    bufferevent_setcb(bev_in, read_client_cb, NULL, event_client_cb,
NULL);

    bufferevent_enable(bev_in, EV_READ);

    return ;
}

void proxy_init()
{
    g_listener = evconnlistener_new_bind(g_base, accept_cb, NULL,
LEV_OPT_REUSEABLE, -1, (struct sockaddr*)&g_local_addr, sizeof(struct
sockaddr_in));
    if ( !g_listener )
    {
        LOG(LOG_LEVEL_FATAL, "Initial g_listener failed.");
    }
    return ;
}
```

All source code of this project is available at <https://github.com/miyatsu/thesis>

## Evaluation data

### Raw data of test

Nginx:

Number of Concurrency	Number of successful access	Number of connection failure	Number of unfinished transition
100	100	0	0
500	341	159	0
1000	525	475	0
2000	963	1037	0
4000	1952	2048	0
6000	2453	3747	0
8000	3067	4933	0
10000	3597	6403	0

Ktrix:

Number of Concurrency	Number of successful access	Number of connection failure	Number of unfinished transition
100	100	0	0
500	482	3	15
1000	658	329	13
2000	1227	747	26
4000	2443	1536	21
6000	3120	2863	17
8000	3536	4437	27
10000	4111	5869	20

### Test program source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <pthread.h>

struct sockaddr_in addr;
```

```
int success = 0;
int failed = 0;
int failed_conn = 0;

char request[] = "GET / HTTP/1.1\r\nHost: 127.0.0.1:8080\r\n\r\n";

void* thread_func(void *arg)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if ( sockfd <= 0 )
    {
        perror("socket error.\n");
        exit(-1);
    }
    int connect_result = connect(sockfd, (struct sockaddr *)&addr,
sizeof(struct sockaddr_in));
    if ( connect_result < 0 )
    {
        goto FAILED_CONN;
    }
    write(sockfd, &request, sizeof(request));
    char str[13] = {0};
    read(sockfd, &str, 13);
    if ( 0 == strncmp(str, "HTTP/1.1 200", 12) )
    {
        goto SUCCESS;
    }
    else
    {
        goto FAILED;
    }
SUCCESS:
    success++;
    close(sockfd);
    return (void *)NULL;
FAILED:
    failed++;
    close(sockfd);
    return (void *)NULL;
FAILED_CONN:
    failed_conn++;
    close(sockfd);
    return (void *)NULL;
}
```

```
int main(int argc, char * argv[])
{
    memset(&addr, 0, sizeof(struct sockaddr_in));
    addr.sin_family = AF_INET;
    if ( 1 != inet_pton(AF_INET, (const char*)argv[1], &addr.sin_addr) )
    {
        printf("address error!\n");
        exit(-1);
    }
    addr.sin_port = htons(8080);

    int n = 0;
    for ( int i = 0; argv[2][i] != '\0'; ++i )
    {
        n += 10;
        n += argv[2][i] - '0';
    }
    pthread_t * tid = malloc(sizeof(pthread_t) * n);
    for ( int i = 0; i < n; ++i )
    {
        pthread_create(&tid[i], NULL, thread_func, NULL);
    }
    sleep(1);
    for ( int i = 0; i < n; ++i )
    {
        void * ret;
        pthread_join(tid[i], &ret);
    }
    printf("\n\nsuccess : %d\nfailed_conn : %d\nfailed : %d\n", success,
failed_conn, failed);
    return 0;
}
```