



北 京 科 技 大 学

计算机网络课程设计报告

学 院： 计算机与通信工程学院

班 级： 计科 184

姓 名： 于永超 任世奇 何思宇 王丹琳

学 号 41821084 41821086 41821094 41824179

成 绩：

指导教师签字：

2021 年 4 月

北京科技大学实验报告

学院：计算机与通信工程学院 专业： 计算机科学与技术 班级： 计科 184

于永超, 任世奇 41821084, 41821086
姓名: 何思宇, 王丹琳 学号: 41821094, 41824179 实验日期: 2021 年 4 月

实验名称：计算机网络课程设计

实验目的：

将书本上抽象的概念与具体实现技术结合，通过网络软件编程的实践，深入理解理论课上学习到的计算机网络基本原理和重要协议，通过自己动手编程封装与发送这些协议的数据包，加深对网络协议的理解，掌握协议传输单元的结构和工作原理及其对协议栈的贡献。

实验仪器：

实验硬件设备：二层交换机一台、腾讯云服务器、

实验软件要求：Ubuntu18.04

小组成员及分工：

于永超，负责报告撰写，实验一代码编写
任世奇，负责报告撰写，实验一二的代码调试
何思宇，负责实验二代码编写及调试
王丹琳，负责报告撰写，实验一代码编写

实验原理：

题目 1 实验原理：

数据包的封装发送和解析（ARP/ICMP/TCP），网络协议栈的多种协议都有自己的功能，协议包括语义、语法和同步三个要素，不同的网络协议其分组的首部格式不同，必须按照协议规定的格式封装（发送）和理解（接收）数据分组首部，才能使得不同站点的计算机按照规定的方式相互通信。ARP 协议是 IP 地址和 MAC 地址解析协议；ICMP 是控制 IP 数据包传递的协议；TCP 是面向连接的可靠的传输层协议，它们均有自己固定的分组首部格式。

本实验中使用 WinPcap 技术或 Socket 技术，根据 ARP/ICMP/TCP 协议数据单元的结构和封装规则，封装数据帧发送到局域网中。另外要捕获网络中的

ARP/ICMP/TCP 数据包，解析数据包的内容，并显示结果，同时写入日志文件。

题目 2 实验原理：

HTTP 协议的作用原理包括四个步骤：

(1)连接：Web 浏览器与 Web 服务器建立连接，打开一个称为 socket（套接字）的虚拟文件，此文件的建立标志着连接建立成功。

(2)请求：Web 浏览器通过 socket 向 Web 服务器提交请求。HTTP 的请求一般是 GET 或 POST 命令（POST 用于 FORM 参数的传递）。

(3)应答：Web 浏览器提交请求后，通过 HTTP 协议传送给 Web 服务器。Web 服务器接到后，进行事务处理，处理结果又通过 HTTP 传回给 Web 浏览器，从而在 Web 浏览器上显示出所请求的页面。

根据上述 HTTP 协议的作用原理，web 服务器主线程监听客户机的连接建立请求，为每次请求主线程监听客户机的连接建立请求，为每次请求/响应创建一个单独的响应创建一个单独的 TCP 连接，一个单独的线程将处理这些连接。并且采用了 epoll 技术，用来解决高并发的场景：

我们通过 `create_epoll(int maxfds)` 来创建一个 epoll 的句柄。之后在网络主循环里面，每一帧的调用 `epoll_wait()` 来查询所有的网络接口，看哪一个可以读，哪一个可以写了。当 `epoll_wait` 这个函数操作成功之后，`epoll_events` 里面将储存所有的读写事件。在循环中，我们通 epoll 的事件注册函数 `int epoll_ctl()` 向 epoll 对象中添加、修改或者删除感兴趣的事件

实验内容与步骤：

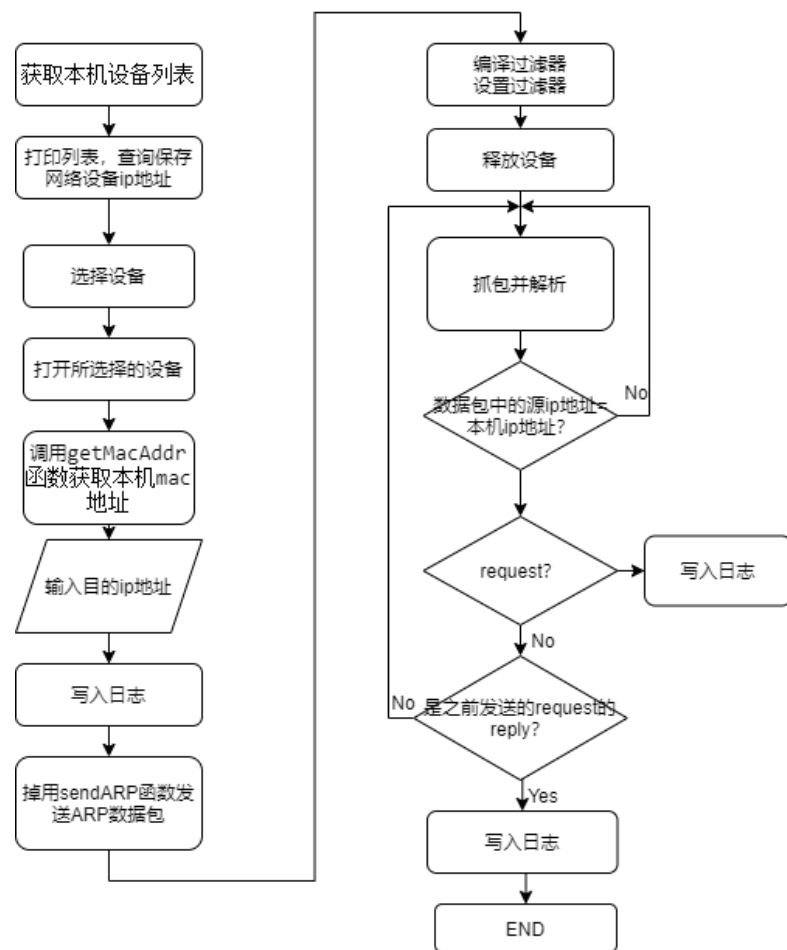
题目 1：数据包的封装发送和解析（ARP/ICMP/TCP）

(1) 实验内容

- 根据 ARP 协议数据的结构，封装成数据帧发送给另一台计算机；
- 捕获网络中包含 ARP 协议数据的数据帧，解析协议数据的内容，并在标准输出中显示报文首部字段的内容，同时写入日志文件。
- 以命令行或图形界面形式运行程序。
- 开启 Wirshark 抓包软件，核对本地计算机发出与收到的数据分组。

(2) 主要步骤（[详细的实验步骤（系统/方法/算法等）](#)，图文结合）

主体函数



获取本机设备列表, 如果获取失败输出 Error in pcap_findalldevs, 并退出程序

```

pcap_if_t* alldevs; //所有网络适配器
pcap_if_t* d; //选中的网络适配器
int inum; //选择网络适配器
int i = 0; //for 循环变量

if (pcap_findalldevs(&alldevs, errbuf) == -1)
{
    fprintf(stderr, "Error in pcap_findalldevs: %s\n", errbuf);
    fprintf(fp, "Error in pcap_findalldevs: %s\n", errbuf);
    exit(1);
}
  
```

打印列表, 同时写入日志文件

```

for (d = alldevs; d; d = d->next)
{
    printf("%d. %s", ++i, d->name);
    fprintf(fp, "%d. %s", i, d->name);
    if (d->description) {
        printf(" (%s)\n", d->description);
        fprintf(fp, " (%s)\n", d->description);
    }
}
  
```

```

    }
    else {
        printf(" (No description available)\n");
        fprintf(fp, " (No description available)\n");
    }
    //查询并保存网络设备的 ip 地址
    char* str = (char*)"0.0.0.0";
    for (pcap_addr_t* a = d->addresses; a; a = a->next) {
        if (a->addr->sa_family == AF_INET) {
            if (a->addr) {
                str = iptos(((struct
sockaddr_in*)a->addr)->sin_addr.s_addr, i);
                break;
            }
        }
    }
    printf("    IP Address: %s\n", str);
    fprintf(fp, "    IP Address: %s\n", str);
}

if (i == 0)
{
    printf("\nNo interfaces found! Make sure WinPcap is installed.\n");
    return -1;
}

```

输入所要选择的设备序号

```

printf("\nEnter the interface number (1-%d):", i);
fprintf(fp, "\nEnter the interface number (1-%d):", i);
scanf_s("%d", &inum);
fprintf(fp, "%d\n", inum);

```

如果输入的序号错误，释放设备列表，结束程序

```

if (inum < 1 || inum > i)
{
    printf("\nInterface number out of range.\n");
    fprintf(fp, "\nInterface number out of range.\n");
    /* 释放设备列表 */
    pcap_freealldevs(alldevs);
    return -1;
}

```

由于 alldevs 是一个链表，故通过 for 循环，跳转到选中的适配器

```

for (d = alldevs, i = 0; i < inum - 1; d = d->next, i++);

```

打开设备，若打开失败，释放设备列表，结束程序

```
        if ((adhandle = pcap_open(d->name,          // 设备名
                                   65536,            // 65535 保证能捕获到不同数据链路层上的每个数据包的全部内容
                                   PCAP_OPENFLAG_PROMISCUOUS, // 混杂模式
                                   1000,              // 读取超时时间
                                   NULL,              // 远程机器验证
                                   errbuf             // 错误缓冲池
                                   )) == NULL)
        {
            fprintf(stderr, "\nUnable to open the adapter. %s is not supported by WinPcap\n", d->name);
            fprintf(fp, "\nUnable to open the adapter. %s is not supported by WinPcap\n", d->name);
            /* 释放设备列表 */
            pcap_freealldevs(alldevs);
            return -1;
        }
```

通过 getMacAddr 函数获取所选的设备 mac 地址，若成功获取则 res=0，反之 res=1

```
int res = getMacAddr(inum);
```

若 getMacAddr 函数未能自动获取到本机 mac 地址，手动输入

```
if (res != 0) {
    printf("Cannot get MAC address automatically, please input MAC address: ");
    fprintf(fp, "Cannot get MAC address automatically, please input MAC address: ");
    u_int temp;
    for (i = 0; i < 6; i++) {
        scanf_s("%d", &temp);
        net_mac_addr[i] = temp;
        fprintf(fp, "%d ", temp);
    }
    fprintf(fp, "\n");
}
```

输入目的 ip 地址

```
printf("Input the IP Address of destination: ");
fprintf(fp, "Input the IP Address of destination: ");
u_int temp;
for (i = 0; i < 4; i++) {
```

```

        scanf_s("%d", &temp);
        dst_ip[i] = temp;
        fprintf(fp, "%d ", temp);
    }
    fprintf(fp, "\n");

```

打印输出适配器的 mac 地址，目的 ip 地址，本机 ip 地址，并写入日志文件

```

printf("\nThe MAC Address of Adapter %d: ", inum);
fprintf(fp, "\nThe MAC Address of Adapter %d: ", inum);
printAddr(net_mac_addr, MACADDR);

printf("The IP Address of Adapter %d: ", inum);
fprintf(fp, "The IP Address of Adapter %d: ", inum);
printAddr(net_ip_addr[inum], IPADDR);
printf("The IP Address of destination: ");
fprintf(fp, "The IP Address of destination: ");
printAddr(dst_ip, IPADDR);

```

通过 sendARP 函数发送 ARP 数据包，若发送成功则 res=0，反之 res=1

```

res = sendARP(net_ip_addr[inum], dst_ip);

if (res == 0) {
    printf("\nSend packet successfully\n\n");
    fprintf(fp, "\nSend packet successfully\n\n");
}
else {
    printf("Failed to send packet due to: %d\n", GetLastError());
    fprintf(fp, "Failed to send packet due to: %d\n", GetLastError());
}

```

过滤器设置

```

netmask
((sockaddr_in*)((d->addresses)->netmask))->sin_addr.S_un.S_addr;
pcap_compile(adhandle, &fcode, filter, 1, netmask);    //编译过滤器
pcap_setfilter(adhandle, &fcode); //设置过滤器

```

释放设备列表并开始抓包

```

pcap_freealldevs(alldevs);

i = 0;

printf("Catching packets...\n\n");
fprintf(fp, "Catching packets...\n\n");

```

抓包并解析，由于 ARP 封装在 MAC 帧，MAC 帧首部占 14 字节，故 arpheader 偏移 14 字节，若所抓取的数据包源 ip 地址不等于本机 ip 地址，则继续抓包，直到抓到源 ip 地址等于本机 ip 地址的数据包，继续解析，若是 request，则写入日志文件并继续抓包，若是之前 request 的 reply 则停止抓包

```
int begin = -1;
//获取数据包并解析
while (res = pcap_next_ex(adhandle, &header, &pkt_data) >= 0) {
    //超时
    if (res == 0) {
        continue;
    }

    //解析 ARP 包，ARP 封装在 MAC 帧，MAC 帧首部占 14 字节
    ArpHeader* arpheader = (ArpHeader*)(pkt_data + 14);
    if (begin != 0) {
        begin = memcmp(net_ip_addr[inum], arpheader->sip,
sizeof(arpheader->sip));
        if (begin != 0) {
            continue;
        }
    }
    //获取时间戳
    local_tv_sec = header->ts.tv_sec;
    ltime = localtime(&local_tv_sec);
    strftime(timestr, sizeof(timestr), "%H:%M:%S", ltime);
    printf("(%)s ", timestr);
    fprintf(fp, "(%)s ", timestr);

    printf("message %d:\n", ++i);
    fprintf(fp, "message %d:\n", i);
    //设置标志，当收到之前发送的 request 的 reply 时结束捕获
    bool ok = false;
    if (arpheader->op == 256) {
        printf("request message.\n");
        fprintf(fp, "request message.\n");
    }
    else {
        printf("reply message.\n");
        fprintf(fp, "reply message.\n");
        //如果当前报文是 reply 报文，则通过比较 ip 来判断是否之前发
        送的 request 对应的 reply
        if (memcmp(arpheader->dip, net_ip_addr[inum],
sizeof(arpheader->dip)) == 0) {
```



```

        memcpy(dst_mac, arpheader->smac, 6);
        ok = true;
    }
}

```

将 ARP 数据包中的信息打印输出并写入日志文件

```

//获取以太网帧长度
printf("ARP packet length: %d\n", header->len);
fprintf(fp, "ARP packet length: %d\n", header->len);

//打印源 mac
printf("source mac: ");
fprintf(fp, "source mac: ");
printAddr(arpheader->smac, MACADDR);
//打印源 ip
printf("source ip: ");
fprintf(fp, "source ip: ");
printAddr(arpheader->sip, IPADDR);
//打印目的 mac
printf("destination mac: ");
fprintf(fp, "destination mac: ");
printAddr(arpheader->dmac, MACADDR);
//打印目的 ip
printf("destination ip: ");
fprintf(fp, "destination ip: ");
printAddr(arpheader->dip, IPADDR);

printf("\n\n");
fprintf(fp, "\n\n");
if (ok) {
    printf("Get the MAC address of destination: ");
    fprintf(fp, "Get the MAC address of destination: ");
    printAddr(dst_mac, MACADDR);
    printf("\nEnd of catching...\n\n");
    fprintf(fp, "\nEnd of catching...\n\n");
    break;
}
}

```

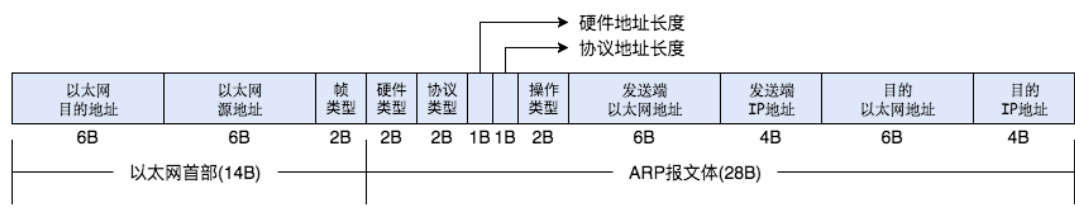
关闭文件流，结束程序

```

fclose(fp);
getchar();
return 0;

```

定义 ARP 协议格式



```
#define ETH_ARP          0x0806  //以太网帧类型表示后面数据的类型，对于 ARP
请求或应答来说，该字段的值为 0x0806
#define HARDWARE        1  //硬件类型字段值为表示以太网地址
#define ETH_IP          0x0800  //协议类型字段表示要映射的协议地址类型值为
0x0800 表示 IP 地址

//14 字节以太网首部
struct EthHeader
{
    u_char DestMAC[6];
    u_char SourMAC[6];
    u_short EthType;
};

//28 字节 ARP 帧结构
struct ArpHeader
{
    unsigned short hdType;
    unsigned short proType;
    unsigned char hdSize;
    unsigned char proSize;
    unsigned short op;
    u_char smac[6];
    u_char sip[4];
    u_char dmac[6];
    u_char dip[4];
};

//定义整个 arp 报文包，总长度 42 字节
struct ArpPacket {
    EthHeader ed;
    ArpHeader ah;
};
```

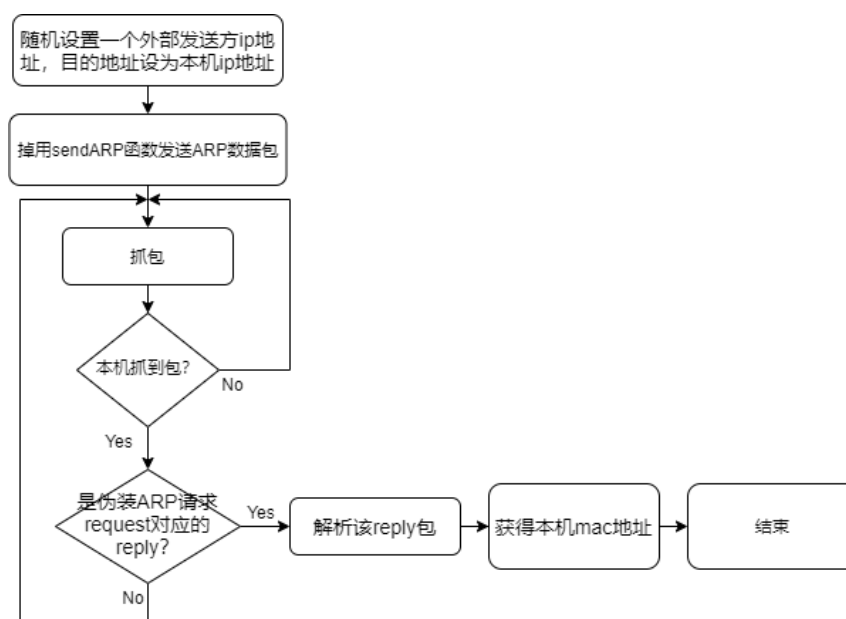
封装 ARP 数据包并广播发送

```

int sendARP(u_char * src_ip, u_char * dst_ip)
{
    unsigned char sendbuf[42]; //arp 包结构大小, 42 个字节
    EthHeader eh;
    ArpHeader ah;
    memcpy(eh.DestMAC, dst_mac, 6);           //以太网首部目的 MAC 地址, 全为
    广播地址
    memcpy(eh.SourMAC, net_mac_addr, 6); //以太网首部源 MAC 地址
    memcpy(ah.smac, net_mac_addr, 6); //ARP 字段源 MAC 地址
    memcpy(ah.dmac, dst_mac, 6);           //ARP 字段目的 MAC 地址
    memcpy(ah.sip, src_ip, 4);             //ARP 字段源 IP 地址
    memcpy(ah.dip, dst_ip, 4);             //ARP 字段目的 IP 地址
    eh.EthType = htons(ETH_ARP);           //htons: 将主机的无符号短整形数
    转换成网络字节顺序
    ah.hdType = htons(HARDWARE);
    ah.proType = htons(ETH_IP);             //上层协议设置为 IP 协议
    ah.hdSize = 6;
    ah.proSize = 4;
    ah.op = htons(REQUEST);
    memset(sendbuf, 0, sizeof(sendbuf)); //ARP 清零
    memcpy(sendbuf, &eh, sizeof(eh));
    memcpy(sendbuf + sizeof(eh), &ah, sizeof(ah));
    return pcap_sendpacket(adhandle, sendbuf, 42); // 发送 ARP 数据包并返回
    发送状态
}

```

通过构造一个外来 ARP 请求获取当前网卡的 MAC 地址



```

int getMacAddr(int curAdapterNo)

```

```

{
    u_char src_ip[4] = { 0x12, 0x34, 0x56, 0x78 }; //随机一个外部发送方的
ip 地址
    u_char dst_ip[4];
    memcpy(dst_ip, net_ip_addr[curAdapterNo], 4); //目的 ip 地址设置为本机
的适配器 id
    memcpy(net_mac_addr, random_mac, 6); //外部发送方的 mac
地址设置为随机的 mac 地址
    int res = sendARP(src_ip, dst_ip);
    if (res != 0) {
        return -1;
    }
    while (res = pcap_next_ex(adhandle, &header, &pkt_data) >= 0) {
        if (res == 0) {
            continue;
        }
        ArpHeader* arph = (ArpHeader*)(pkt_data + 14);
        if (arph->op != 256) {
            if (memcmp(arph->dip, src_ip, sizeof(src_ip)) == 0) { //
收到了伪装 ARP 请求 request 对应的 reply, 解析该 reply 包获得本机 mac 地址
                memcpy(net_mac_addr, arph->smac, 6);
                break;
            }
        }
    }
    return 0;
}

```

题目 2: 多线程 Web 服务器

(1) 实验内容

设计并实现一个 WEB 服务器, 可并行服务处理多个请求, 实现 HTTP1.0 的功能。在后台运行服务器程序。

使用 Socket API 或 WinPcap 技术。

主线程监听客户机的连接建立请求, 为每次请求主线程监听客户机的连接建立请求, 为每次请求/响应创建一个单独的响应创建一个单独的 TCP 连接, 一个单独的线程将处理这些连接。

并且采用了 epoll 技术，用来解决高并发的场景：

在 linux 的网络编程中，很长的时间都在使用 select 来做事件触发。在 linux 新的内核中，有了一种替换它的机制，就是 epoll。相比于 select，epoll 最大的好处在于它不会随着监听 fd 数目的增长而降低效。

我们通过 `create_epoll(int maxfds)` 来创建一个 epoll 的句柄，其中 `maxfds` 为 epoll 所支持的最大句柄数。这个函数会返回一个新的 epoll 句柄，之后的所有操作将通过这个句柄来进行操作。之后在网络主循环里面，每一帧的调用 `epoll_wait(int epfd, epoll_event events, int max events, int timeout)` 来查询所有的网络接口，看哪一个可以读，哪一个可以写了。当 `epoll_wait` 这个函数操作成功之后，`epoll_events` 里面将储存所有的读写事件。`max_events` 是当前需要监听的所有 socket 句柄数。最后一个 `timeout` 是 `epoll_wait` 的超时，为 0 的时候表示马上返回，为 -1 的时候表示一直等下去，直到有事件范围，为任意正整数的时候表示等这么长的时间。

在循环中，我们通 epoll 的事件注册函数 `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)` 向 epoll 对象中添加、修改或者删除感兴趣的事件。`epoll_ctl()` 不同于 `select()` 是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。`epoll_wait` 方法返回的事件必然是通过 `epoll_ctl` 添加到 epoll 中的。


用浏览器打开，显示请求页面内容即可；如有差错，则显示出错信息。

(2) 主要步骤

项目目录结构及注解如下：

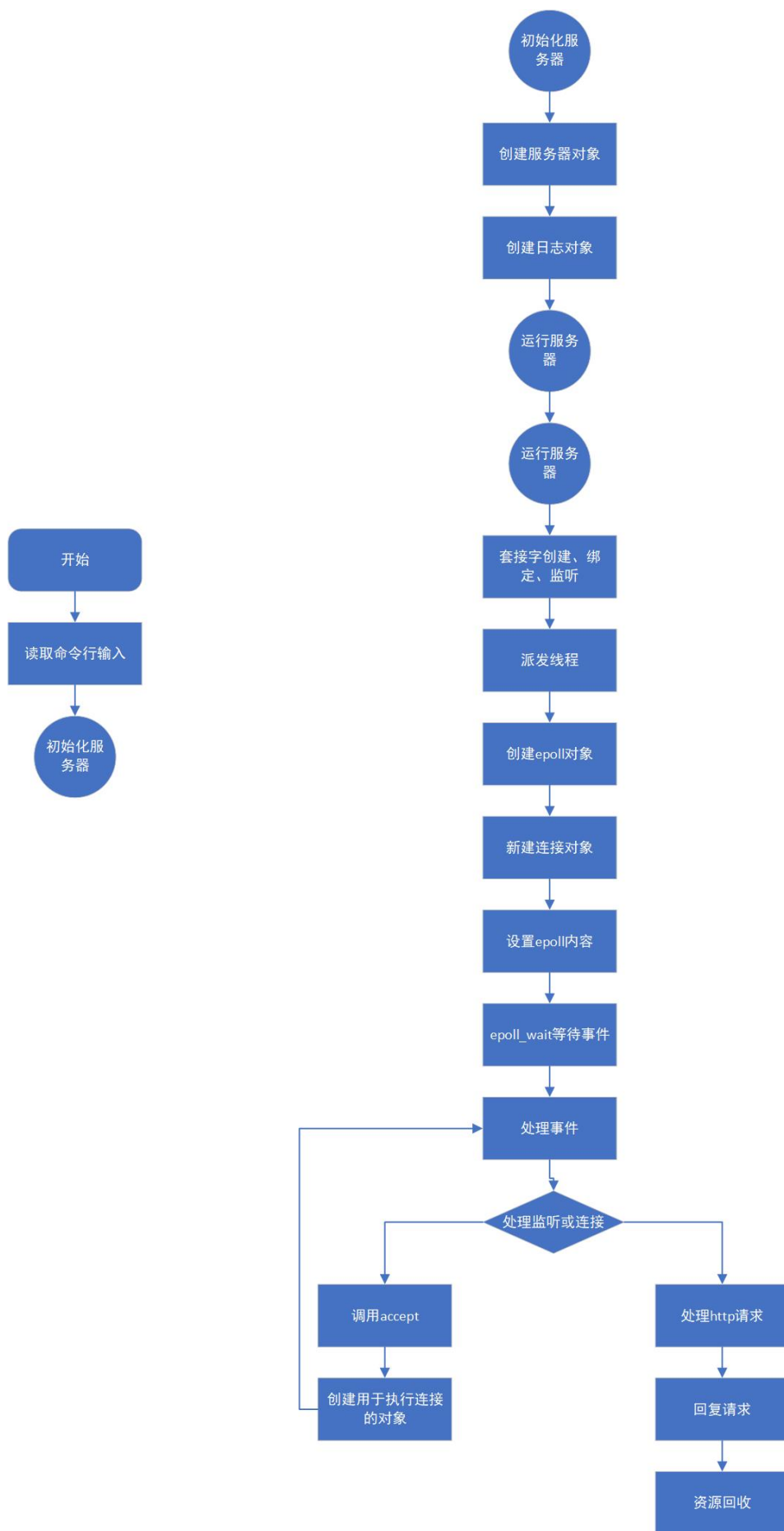
```
web
├── 'async' $' \r'
├── build.sh 编译脚本
├── home
│   ├── logs 存放日志文件
│   └── webhome 存放静态资源文件
│       ├── 404.html
│       ├── img
│       │   └── title.png
│       ├── index.html
│       ├── news.html
│       └── test.html
└── http 存放处理 http 请求的相关函数文件
```

```
|   ├── http.cpp
|   ├── http.h
|   ├── http_error.h
|   ├── http_util.cpp
|   ├── http_util.h
|   └── SConscript.py
└── lib 存放依赖文件
    ├── buffer.cpp
    ├── buffer.h
    ├── buffer.hpp
    ├── daemon.c
    ├── daemon.h
    ├── fileio.c
    ├── fileio.h
    ├── hsy_flock.h
    ├── hsy_path.h
    ├── hsy_process.h
    ├── hsy_string.h
    ├── hsy_time.h
    ├── list.cpp
    ├── list.h
    ├── max_heap.h
    ├── path.cpp
    ├── SConscript.py
    ├── serror.h
    ├── socket.c
    └── socket.h
└── log 存放生成日志的相关函数文件
    ├── log.cpp
    ├── log.h
    └── SConscript.py
└── rbtree 存放红黑树相关函数文件
    ├── rbtree.cpp
    ├── rbtree.h
    └── SConscript.py
└── server 存放服务器运行相关函数文件
    ├── async
    ├── async_handler.cpp
    ├── async_handler.h
    ├── async_server.cpp
    ├── async_server.h
    ├── conn_ctx.h
    ├── epoll.cpp
    └── epoll.h
```



```
├─ main.cpp
└─ SConstruct.py
```

运行流程图如下：



server\main.cpp

程序入口，设置主机地址、端口号、home 文件目录、线程数，并调用初始化服务器函数和启动服务器函数。

```
int main(int argc ,char * argv[])
{
    bool is_nostub = false;
    int port = 8989,ret,thread_num = 0;
    char * host = "82.156.86.176", *home = "home";

    port = atoi(argv[1]);
    std::cout << "host:" << host << ",port:" << port << ",home:" << home <<
    ",thread_num:" <<thread_num <<"\n";

    //init the server including socket bind and listen
    if(init_async_server(is_nostub,home,thread_num)){
        perror("init server error\n");
        return EXIT_FAILURE;
    }
    std::cout << "init server ok\n";

    //run the server
    if(start_async_server(port)){
        perror("start server error\n");
        ret = EXIT_FAILURE;
        if (cleanup_async_server()) {
            perror("clean up server error\n");
            return EXIT_FAILURE;
        }

        return ret;
    }
    ret = EXIT_SUCCESS;

    if(cleanup_async_server()){
        perror("clean up server error\n");
        return EXIT_FAILURE;
    }

    return ret;
}
```

server\async_server.cpp

在此处进行初始化服务器和启动服务器的相关操作。初始化服务器主要工

作是创建 Async_server_ctx 对象和日志对象，其中包含了连接超时、epoll 超时、日志文件指针等常量的设置。启动服务器的主要工作是派发监听线程和处理线程，主要使用自行定义的 lx_listen()、do_service() 函数和 pthread_create()、pthread_detach() 来完成。

```
int init_async_server(bool is_nostub, const char * home,int thread_num)
{
    char buff[1024];
    int ret = 0;
    global_ctx = new Async_server_ctx(12000, 12000, 200, is_nostub);
    if(global_ctx == NULL)
    {
        perror("malloc in init error\n");
        return -1;
    }

    if(home)
        g_home = (char *)home;

    if(snprintf(buff,1024,"%s/%s",g_home, g_loghome) <=0) {
        perror("snprintf log path error");
        pthread_mutex_destroy(&global_ctx->getmutex());
        if (global_ctx != NULL)
            delete(global_ctx);
        global_ctx = NULL;
    }

    global_ctx->setlog(new log::Log(buff, "access.log", LX_LOG_DEBUG, 1, 1,
0));
    global_ctx->getlog()->setdailyarg(global_ctx->getasarg());
    global_ctx->getlog()->setarg(&(global_ctx->getasarg()));
    global_ctx->getlog()->setplock(0);
    global_ctx->getlog()->settlock(1);
    if ( thread_num <= 0 && (thread_num = sysconf(_SC_NPROCESSORS_ONLN) ) <
0) {

        perror("get cpu core number error");
        ret = -1;
        global_ctx->getlog()->cleanup();
        pthread_mutex_destroy(&global_ctx->getmutex());
        if (global_ctx != NULL)
            delete(global_ctx);
        global_ctx = NULL;
        return -1;
    }
```

```

    }
    global_ctx->setthreadnum(thread_num);

    return 0;
}

int start_async_server(int port)
{
    int i,ret,listen_fd ;
    pthread_t tid;
    if( (listen_fd = lx_listen(port)) < 0 ){
        global_ctx->getlog()->logerror("lx_listen          error[%d:%s]",ret,
strerror(ret));
        return -1;
    }
    for(i = 0; i < global_ctx->getthreadnum(); ++i){
        if( ret = pthread_create(&tid, NULL,do_service,(void *)listen_fd )){
            global_ctx->getlog()->logerror("pthread_create
error[%d:%s]",ret, strerror(ret));
            return -1;
        }
        if( ret = pthread_detach(tid)){
            global_ctx->getlog()->logerror("pthread_detach
error[%d:%s]",ret, strerror(ret));
            return -1;
        }

        global_ctx->getlog()->loginfo("server %ld start",(long)tid);
    }

    global_ctx->getlog()->loginfo("Starting server succeed. The number of
working threads is %d",global_ctx->getthreadnum());

    while (1)
        sleep(5);

    if(listen_fd >=0)
        close(listen_fd);

    return 0;
}

```

lib\socket.c

lx_listen() 函数的主要工作是用 socket(), bind(), listen() 函数，建立套接口，绑定套接口并创建监听套接口，返回值是套接口的文件描述符。

```
int lx_listen( short port)
{
    const int on = 1;
    int listen_fd = -1;
    struct sockaddr_in addr;
    if(lx_block_sigpipe())
    {
        if (listen_fd >= 0)
            close(listen_fd);
        return -1;
    }

    listen_fd = socket(PF_INET, SOCK_STREAM, 0);
    if(listen_fd == -1)
    {
        if (listen_fd >= 0)
            close(listen_fd);
        return -1;
    }

    if(setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)))
    {
        if (listen_fd >= 0)
            close(listen_fd);
        return -1;
    }

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = PF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(listen_fd, (struct sockaddr *)&addr, sizeof(addr)))
    {
        if (listen_fd >= 0)
            close(listen_fd);
        return -1;
    }

    if (listen(listen_fd, SOMAXCONN))
    {
        if (listen_fd >= 0)
```

```

        close(listen_fd);
    return -1;
}
return listen_fd;

}

```

server\async_server.cpp

do_service()函数主要功能是调用 epoll_create()创建 epoll 对象，新建 Async_conn_ctx 连接对象（其中包括 listen_handle 实现 accept()），调用 epoll_wait()函数等待事件，并完成对监听到事件的处理。这里我们使用了红黑树来实现定时器。

```

void* do_service(void * arg)
{
    int ret = 0, ep_fd = -1,listen_fd = -1;
    struct epoll_event *events = NULL;
    Async_conn_ctx* listen_ctx = NULL;
    rbtree::RbTree_t timer;
    //timer.root = new rbtree::RbTreeNode();
    //timer.nil = *(timer.root);
    timer.nil = *(new rbtree::RbTreeNode());
    timer.root = &(timer.nil);
    //lx_rbtree_init(&timer,malloc,free);
    listen_fd = (int)(long)arg;

    if( (ep_fd = epoll_create(20)) < 0 ){
        global_ctx->getlog()->logerror("epoll_create error");
        return NULL;
    }

    if( (events = (struct epoll_event *)malloc(global_ctx->getmaxevent() *
sizeof(struct epoll_event) ) )
        == NULL) {
        global_ctx->getlog()->logerror("malloc epoll events error");
        ret = -1;
        if (timer.root)
        {
            delete(timer.root);
            timer.root = nullptr;
        }
        if (ep_fd >= 0)
            close(ep_fd);

        if (events)

```

```

        free(events);

    if (listen_ctx)
    {
        delete(listen_ctx);
        listen_ctx = nullptr;
    }
    return NULL;
}

    listen_ctx    =    new    Async_conn_ctx(ep_fd,    listen_fd,    &timer,
listen_handle);
    listen_ctx->setstage(STAGE_LISTENING);

    if(    lx_set_epoll(ep_fd,listen_fd,listen_ctx,EPLLIN|EPOLLET,    true,
true)){
        global_ctx->getlog()->logerror("lx_set_epoll error" );
        ret = -1;
        if (timer.root)
        {
            delete(timer.root);
            timer.root = nullptr;
        }
        if (ep_fd >= 0)
            close(ep_fd);

        if (events)
            free(events);

        if (listen_ctx)
        {
            delete(listen_ctx);
            listen_ctx = nullptr;
        }
        return NULL;
    }

    for(;;){
        ret                                =                                epoll_wait(ep_fd,events,
global_ctx->getmaxevent(),global_ctx->gettimeout());
        if(ret < 0){
            if(errno != EINTR){
                global_ctx->getlog()->logerror("epoll_wait error");
                ret = -1;
            }
        }
    }
}

```

```

        break;
    }else
        continue;
    }else if(ret > 0)
    //      std::cout << "doing handle_events" << std::endl;
        handle_events(ep_fd, events, ret, &timer);
}
if(ret != -1)
    ret = 0;

if(timer.root)
{
    delete(timer.root);
    timer.root = nullptr;
}
if(ep_fd >=0)
    close(ep_fd);

if(events)
    free(events);

if (listen_ctx)
{
    delete(listen_ctx);
    listen_ctx = nullptr;
}
return NULL;
}

```

server\async_handler.cpp

handle_events() 函数的功能是利用红黑树管理连接对象，包括删除超时的连接。

listen_handle() 函数中调用 accept(), 并创建了新的连接对象。

conn_handle() 函数对请求进行处理，连接对象有不同的 stage，包括 STAGE_START, STAGE_REQ_HEAD, STAGE_REQ_BODY, STAGE_RESP_HEAD, STAGE_RESP_BODY，具体实现细节在 req_head_handle(), req_body_handle(), resp_head_handle(), resp_body_handle() 中。

```

int handle_events(int ep_fd, struct epoll_event * events, int nevent,
rbtree::RbTree_t* timer)
{
    int i = 0, ret = 0;
    uint64_t key_temp;
    char ebuff[1024] ,*pebuff;
    Async_conn_ctx* cctx;

```

```

rbtree::RbTreeNode * n;
//if (nevent) printf("handle events ,nevent :%d\n",nevent);
std::cout << "nevent:"<<nevent<<std::endl;
for(; i < nevent;++i) {
    cctx = (Async_conn_ctx*)events[i].data.ptr;
    cctx->setevents(events[i].events);

    pebuff = lx_get_events(cctx->getevents(),ebuff,1024 );
    //printf("events flags:%s\n",pebuff);

    ret = cctx->handle(cctx);
}

key_temp = (uint64_t)get_micros();
while(1)
{
    n = rbtree::rbTree_min(timer,timer->root);
    if(n != nullptr && n != &timer->nil )
    {

        if(n->getkey() == 0)
        {
            std::cout << "you\n";
        }

        if (n->getkey() < key_temp)
        {
            uint64_t key = n->getkey();

            global_ctx->getlog()->loginfo("timer:connection timeout");
            //printf("delete key in timer check:%lu\n",(unsigned
long)n->key);

            remove_conn((Async_conn_ctx*)n->getdata());
            if(rbtree::rbTreeNode_delete(timer,n->getkey()) != 0)
            {
                global_ctx->getlog()->logfatal("timer: cannot find
key %lu for delete",(unsigned long) key);
            }

        }
        else
        {

```



```

        break;
    }
}
else
{
    break;
}
}
return 0;
}

int listen_handle(Async_conn_ctx* arg)
{
    int ret = 0, fd = -1;
    socklen_t addrlen = 0;
    struct sockaddr_in addr;
    Async_conn_ctx* cctx = NULL;
    uint64_t l_temp;
    if(arg->getevents() & EPOLLERR ) {
        global_ctx->getlog()->logerror("listen fd from epoll error");
        return 0;
    }

    addrlen = sizeof(struct sockaddr_in);
    if(pthread_mutex_trylock(&global_ctx->getmutex()) == 0) {
        std::cout<< "start to accept"<<std::endl;
        fd = accept(arg->getfd(), (struct sockaddr *)&addr, &addrlen);
        if(pthread_mutex_unlock(&global_ctx->getmutex())) {
            global_ctx->getlog()->logerror("%s:pthread_mutex_unlock
error", __FUNCTION__);
            ret = -1;
            if (cctx)
                delete(cctx);

            if (fd >= 0) {
                close(fd);
            }
            return ret;
        }
    }else{
        return 0;
    }

    if(fd < 0) {

```

```

        if(errno == EAGAIN|| errno == EWOULDBLOCK) {
            //g_ctx->log.logdebug(&g_ctx->log,"accept return again or
block,%d:%s",errno,strerror(errno));
            return 0;
        }else{
            global_ctx->getlog()->logerror("accept error");
            return 0;
        }
    }
    cctx = new Async_conn_ctx(arg->getepfd(), fd, arg->gettimer(), addr,
conn_handle);

if(lx_set_epoll(arg->getepfd() ,cctx->getfd(),cctx,EPOLLIN|EPOLLET,true,true)){
    global_ctx->getlog()->logerror("set_epoll error");
    ret = -1;
    if (cctx)
        delete(cctx);

    if (fd >= 0) {
        close(fd);
    }
    return ret;
}
l_temp = get_micros() + global_ctx->getconntimeout()*1000;
std::cout << "the timeout of new node = " << l_temp << std::endl;
while(rbtree::rbTreeNode_insert(arg->gettimer(),(uint64_t)l_temp,
cctx) != 0){
    l_temp += 1;
    std::cout << "l_temp + 1 succeed"<< std::endl;
}
cctx->settimeout(l_temp);
//printf("insert key:%lu\n", (unsigned long)l_temp);

printf("listen handle call end\n");
return 0;

}

int conn_handle(Async_conn_ctx* arg)
{
    int ret;
    rbtree::RbTree_t* root;
    uint64_t key;

```

```

        if(arg->getevents() & EPOLLERR || arg->getevents() & EPOLLHUP) {
            global_ctx->getlog()->logerror("%s:connection error
occur",__FUNCTION__);
            root = arg->gettimer();
            key = (uint64_t)arg->gettimeout();
            ret = rbtree::rbTreeNode_delete(root, key);
            remove_conn(arg);
            arg = nullptr;
            return 0;
        }
        switch(arg->getstage()) {

        case STAGE_START:
            arg->setstage(STAGE_REQ_HEAD);

        case STAGE_REQ_HEAD:
            if ((ret = req_head_handle(arg)) == HANDLE_NEED_MORE)
            {
                return 0;
            }
            else if(ret == HANDLE_ERR)
            {
                root = arg->gettimer();
                key = (uint64_t)arg->gettimeout();
                ret = rbtree::rbTreeNode_delete(root, key);
                remove_conn(arg);
                arg = nullptr;
                return 0;
            }
            arg->setstage(STAGE_REQ_BODY);
            arg->setcontlen(0);
            arg->setinoutlen(0);

            /*
            if(http_print_http(&(*this)._req_ctx)) {
                g_ctx->log.logerror(&g_ctx->log,"print_pare_info error");
            }

            */

        case STAGE_REQ_BODY:
            if ((ret = req_body_handle(arg)) == HANDLE_NEED_MORE)
            {
                return 0;
            }

```

```

else if (ret == HANDLE_ERR)
{
    root = arg->gettimer();
    key = (uint64_t)arg->gettimeout();
    ret = rbtree::rbTreeNode_delete(root, key);
    remove_conn(arg);
    arg = nullptr;
    return 0;
}

arg->setstage(STAGE_RESP_HEAD);
arg->getdatabuff()->setoffset(0);
arg->getdatabuff()->setlen(0);
if (lx_set_epoll(arg->getepfd(), arg->getfd(), arg,
EPOLLOUT|EPOLLET, true, false)) {
    global_ctx->getlog()->logerror("%s:lx_set_epoll
error", __FUNCTION__);
    root = arg->gettimer();
    key = (uint64_t)arg->gettimeout();
    ret = rbtree::rbTreeNode_delete(root, key);
    remove_conn(arg);
    arg = nullptr;
    return 0;
}

return 0;

case STAGE_RESP_HEAD:
    if ((ret = resp_head_handle(arg)) == HANDLE_NEED_MORE)
    {
        return 0;
    }
    else if (ret == HANDLE_ERR)
    {
        root = arg->gettimer();
        key = (uint64_t)arg->gettimeout();
        ret = rbtree::rbTreeNode_delete(root, key);
        remove_conn(arg);
        arg = nullptr;
        return 0;
    }
    arg->setstage(STAGE_RESP_BODY);
    arg->setfh(NULL);

case STAGE_RESP_BODY:
    if ((ret = resp_body_handle(arg)) == HANDLE_NEED_MORE)

```

```

    {
        return 0;
    }
    else if (ret == HANDLE_ERR)
    {
        root = arg->gettimer();
        key = (uint64_t)arg->gettimeout();
        ret = rbtree::rbTreeNode_delete(root, key);
        remove_conn(arg);
        arg = nullptr;
        return 0;
    }
    arg->setstage(STAGE_DONE);
    record_end_log(arg);
default:
    root = arg->gettimer();
    key = (uint64_t)arg->gettimeout();
    ret = rbtree::rbTreeNode_delete(root, key);
    remove_conn(arg);
    arg = nullptr;
    return 0;
}
//std::cout << "this conn step has finished" << std::endl;
return 0;
}

```

server\async_handler.cpp

req_head_handle() 对请求头进行处理。主要用到了 http_parse() 函数。

```

int req_head_handle(Async_conn_ctx* arg)
{
    int ret = 0, read_num = 0, to_read_num = 0;
    char * buff;
    while(1)
    {
        buff = lx_buffer_lenp((arg->getrequest()->getoriginbuff()));
        to_read_num
lx_buffer_freenum((arg->getrequest()->getoriginbuff()));
        read_num = recv(arg->getfd(), buff, to_read_num, 0);
        if(read_num < 0)
        {
            if(AGAIN_FLAG)
            {
                return HANDLE_NEED_MORE;
            }
        }
    }
}

```

```

        }
        else
        {
            global_ctx->getlog()->logerror("%s:recv
error",__FUNCTION__);
            return HANDLE_ERR;
        }
    }
    else if(read_num == 0)
    {
        global_ctx->getlog()->logerror("cannot get enough head
info");
        return HANDLE_ERR;
    }
    else
    {
        arg->getrequest()->getoriginbuff()->setlen(arg->getrequest()->getoriginbuff()->
getlen() + read_num);
        ret = http_parse(arg->getrequest());
        if( ret == HEC_OK)
        {
            //copy to data_buff
            int tocopy =
lx_buffer_unscannum(arg->getrequest()->getoriginbuff());
            if(tocopy >= (arg->getdatabuff()->getmaxlen())) {
                global_ctx->getlog()->logerror("req body too big to copy
to data_buff");
                return HANDLE_ERR;
            }
            Buffer* tmp = arg->getrequest()->getoriginbuff();
memcpy(arg->getdatabuff()->getbase(), lx_buffer_offsetp(tmp), tocopy);
            arg->getdatabuff()->setlen(tocopy);

            return HANDLE_DONE;
        }
        else if( ret == HEC_NEED_MORE)
            continue;
        else
        {
            global_ctx->getlog()->logerror("parser error[%d]",ret);

```

```

        return HANDLE_ERR;
    }
}

return 0;
}

```

server\async_handler.cpp

req_body_handle ()对请求体进行处理。主要用到了 http_get_contlen ()函数。

```

int req_body_handle(Async_conn_ctx* arg)
{
    int nleft,to_read_num,read_num ;
    Buffer *data;

    data = arg->getdatabuff();
    arg->setcontlen(http_get_contlen(arg->getrequest()->getinfo()));

    if(arg->getcontlen() <= 0 )
        return HANDLE_DONE;

    if(arg->getinoutlen() == 0){
        arg->setinoutlen(arg->getcontlen() > data->getlen() ?
data->getlen() : arg->getcontlen());
    }

    while(arg->getinoutlen() < arg->getcontlen()){

        nleft = arg->getcontlen() - arg->getinoutlen();
        to_read_num = nleft < data->getmaxlen()? nleft : data->getmaxlen();
        read_num = recv(arg->getfd(),data->getbase(), to_read_num,0);
        if(read_num < 0){
            if(AGAIN_FLAG ){
                return HANDLE_NEED_MORE;

            }else{
                global_ctx->getlog()->logerror("%s:recv err",__FUNCTION__);
                return HANDLE_ERR;

            }
        }else if(read_num == 0){
            global_ctx->getlog()->logerror("%s:can not get enough req
body",__FUNCTION__);

```

```

        return HANDLE_ERR;

    }else{
        std::cout << data->getbase()<< std::endl;
        arg->setinoutlen(read_num + arg->getinoutlen());
    }
}

return HANDLE_DONE;

}

```

server\async_handler.cpp

resp_head_handle () 对响应头进行处理。主要用到了 http_set_prop1() http_set_rcode(), http_set_headers(), http_set_prop3(), http_seri_head() 以及 send() 函数。

```

int resp_head_handle(Async_conn_ctx* arg)
{
    int ret, send_num, to_send_num;
    int rcode;
    char date[64] ,* uri,*rstr;
    Parser_ctx * response_ctx;
    int temp_num;
    char * headers[] = {
        "Content-Type" , "text/html",
        "Connection"   , "Keep-Alive",
        "Server"        , "ssss/spl 1.0",
        NULL, NULL
    };
    KVListNode* tmp = arg->getresponse()->getinfo()->getheaders();
    response_ctx = arg->getresponse();
    if(arg->getdatabuff()->getlen() == 0){
        if( !(ret = get_browser_time(time(NULL), date, 64))) {
            global_ctx->getlog()->logerror("%s:snprintf          date
error", __FUNCTION__);
            return HANDLE_ERR;
        }

        uri      =      arg->getrequest()->getinfo()->geturi()->getbase()      ?
arg->getrequest()->getinfo()->geturi()->getbase()      :
arg->getrequest()->getinfo()->getbase()      +
arg->getrequest()->getinfo()->geturi()->getoffset();
        if( uri == NULL || strcmp(uri, "/") == 0)
            uri = "/index.html";
    }
}

```



```

        if( (ret = snprintf(arg->getpath(), MAX_PATH_LEN, "%s/%s%s", g_home,
g_whoame, uri)) <= 0 ){
            global_ctx->getlog()->logerror("%s:snprintf          path
error",__FUNCTION__);
            return HANDLE_ERR;

        }

        if( (temp_num = lx_get_fsize(arg->getpath()) )== -1){
            arg->setcontlen(temp_num);
            rcode = 404;
            rstr = "File Not Found";

            global_ctx->getlog()->logerror("uri invalid ,404,uri:%s", uri);

            if( (ret = snprintf(arg->getpath(), MAX_PATH_LEN, "%s/%s%s",
g_home, g_whoame, "/404.html")) <= 0 ){
                global_ctx->getlog()->logerror("%s:snprintf path error,ret
= %d",__FUNCTION__, ret);
                return HANDLE_ERR;

            }

            if( (temp_num = lx_get_fsize(arg->getpath()))== -1){
                arg->setcontlen(temp_num);
                global_ctx->getlog()->logerror("open 404 file error:%s",
arg->getpath());
                return HANDLE_ERR;

            }

            arg->setcontlen(temp_num);
        }else{
            arg->setcontlen(temp_num);
            rcode = RESP_OK;
            rstr = NULL;
        }

        int tmp_int = arg->getresponse()->getinfo()->getprot();
        Buffer* tmp_prot = arg->getresponse()->getinfo()->getprotstr();
        if(arg->getresponse()->http_set_prop1(P_HTTP_1_1,          h_prot_str,
&tmp_int, tmp_prot))
        {
            global_ctx->getlog()->logerror("set prot error");
            return HANDLE_ERR;
        }

```

```

        if( http_set_rcode(arg->getresponse(), rcode, rstr))
        {
            global_ctx->getlog()->logerror("set resp code error");
            return HANDLE_ERR;

        }

        if(arg->getresponse()->http_set_headers(headers,
arg->getcontlen()))
        {
            global_ctx->getlog()->logerror("set headers error");
            return HANDLE_ERR;

        }

        if(response_ctx->http_set_prop3(&tmp, "Date", date, true)){
            global_ctx->getlog()->logerror("set headers error");
            return HANDLE_ERR;

        }

        arg->getdatabuff()->setlen(http_seri_head(response_ctx->getinfo(),
T_RESP, arg->getdatabuff()->getbase(), arg->getdatabuff()->getmaxlen()));

    }

    while( lx_buffer_unscannum(arg->getdatabuff()) > 0 ){
        to_send_num = lx_buffer_unscannum(arg->getdatabuff());
        send_num = send(arg->getfd(), lx_buffer_offsetp(arg->getdatabuff()),
to_send_num, 0);
        if(send_num < 0){
            if( AGAIN_FLAG ){
                return HANDLE_NEED_MORE;

            }else{
                global_ctx->getlog()->logerror("%s:send          error",
__FUNCTION__);
                return HANDLE_ERR;
            }
        }else{
            arg->getdatabuff()->setoffset(arg->getdatabuff()->getoffset()+
send_num);
            continue;
        }
    }
}

```

```

        return HANDLE_DONE;

    }

```

server\async_handler.cpp

resp_body_handle () 对响应体进行处理。主要用到了 fill_send_buff() 和 send() 函数。

```

int resp_body_handle(Async_conn_ctx* arg)
{
    int ret = HANDLE_DONE;
    FILE* tmp;
    if(arg->getfh() == NULL) {
        if ((tmp = fopen(arg->getpath(), "rb")) == NULL) {
            std::cout << "fh is null and open fail" << std::endl;
            arg->setfh(tmp);
            global_ctx->getlog()->logerror("%s:open    file    %s    error",
__FUNCTION__, arg->getpath());
            if (arg->getfh() != NULL) {
                fclose(arg->getfh());
                arg->setfh(NULL);
            }
            return HANDLE_ERR;
        }
        arg->setfh(tmp);
        arg->getdatabuff()->setoffset(0);
        arg->getdatabuff()->setlen(0);
        arg->setinoutlen(0);
        if(fill_send_buff(arg)) {
            global_ctx->getlog()->logerror("%s:fill_send_buff    first
error %s ", __FUNCTION__, arg->getpath());
            if (arg->getfh() != NULL) {
                fclose(arg->getfh());
                arg->setfh(NULL);
            }
            return HANDLE_ERR;
        }
        arg->setinoutlen(arg->getinoutlen()
arg->getdatabuff()->getlen());
    }
    while(1) {
        int send_num, to_send_num;

```

```

to_send_num = lx_buffer_unscannum(arg->getdatabuff());
send_num = send(arg->getfd(), lx_buffer_offsetp(arg->getdatabuff()),
to_send_num, 0);
    if( send_num < 0) {
        if(AGAIN_FLAG) {
            return HANDLE_NEED_MORE;
        }else{
            global_ctx->getlog()->logerror("%s:send      response      body
error", __FUNCTION__);
            if (arg->getfh() != NULL) {
                fclose(arg->getfh());
                arg->setfh(NULL);
            }
            return HANDLE_ERR;
        }
    }else if(send_num == 0) {
        global_ctx->getlog()->logerror("%s:send      response      body
error, send_num = 0", __FUNCTION__);
        if (arg->getfh() != NULL) {
            fclose(arg->getfh());
            arg->setfh(NULL);
        }
        return HANDLE_ERR;
    }
    else
    {
        arg->getdatabuff()->setoffset(arg->getdatabuff()->getoffset() +
send_num);

        if(arg->getdatabuff()->getoffset() ==
arg->getdatabuff()->getlen()) {
            if( arg->getinoutlen() == arg->getcontlen()) {
                if (arg->getfh() != NULL) {
                    fclose(arg->getfh());
                    arg->setfh(NULL);
                }
                return HANDLE_DONE;
            }else{
                if(fill_send_buff(arg)) {
                    global_ctx->getlog()->logerror("%s:fill_send_buff
second error %s ", __FUNCTION__, arg->getpath());
                    if (arg->getfh() != NULL) {

```

```

        fclose(arg->getfh());
        arg->setfh(NULL);
    }
    return HANDLE_ERR;
}
arg->setinoutlen(arg->getdatabuff()->getlen()
arg->getinoutlen());
    continue;
}
} else {
    continue;
}
}
} //end send loop

if(arg->getfh() != NULL) {
    fclose(arg->getfh());
    arg->setfh(NULL);
}
return HANDLE_DONE;
}

```

log\log.cpp

服务器中包含日志系统，在服务器对象创建后即创建日志对象，在服务器运行期间，通过在各处使用 log 对象添加日志，并存放到 logs 文件夹下。效果如下：

```

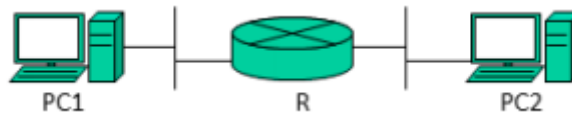
home > logs > access.log.210408_143812
1 2021-04-08_14:38:12 [ ] 20123 server 140675782493952 start
2 2021-04-08_14:38:12 [ ] 20123 Starting server succeed. The number of working threads is 1
3 2021-04-08_14:38:20 [ ] 20123 conn_handle:connection error occur( 11: )
4 2021-04-08_14:38:26 [ ] 20123 uri:/img/title.png,addr:0.0.0.0,start:2021-04-08_14:38:26,duration:6565979
5 2021-04-08_14:38:42 [ ] 20123 uri:/index.html,addr:0.0.0.0,start:2021-04-08_14:38:42,duration:110
6 2021-04-08_14:39:09 [ ] 20123 uri:/index.html,addr:0.0.0.0,start:2021-04-08_14:39:09,duration:110
7 2021-04-08_14:39:20 [p/] 20123 uri invalid ,404,uri:/40402302( 2: )
8 2021-04-08_14:39:20 [ ] 20123 uri:/40402302,addr:0.0.0.0,start:2021-04-08_14:39:20,duration:266
9 2021-04-08_14:39:55 [ ] 20123 uri:/img/title.png,addr:0.0.0.0,start:2021-04-08_14:39:55,duration:6732432
10

```

实验结果与分析：

（详细的实验结果，图/表/文相结合，并对实验结果进行较全面的对比分析，类似于“验证问题”）

1) 实验一



我们将 PC1 和 PC2 连接至同一交换机上，并设置如下 IP：

PC1: 123.123.110.110

PC2: 123.123.110.101

在 PC1 上 ping PC2，通信正常：

```

C:\Users\Administrator>ping 123.123.110.101

正在 Ping 123.123.110.101 具有 32 字节的数据:
来自 123.123.110.101 的回复: 字节=32 时间=1ms TTL=128
来自 123.123.110.101 的回复: 字节=32 时间<1ms TTL=128
来自 123.123.110.101 的回复: 字节=32 时间<1ms TTL=128
来自 123.123.110.101 的回复: 字节=32 时间<1ms TTL=128

123.123.110.101 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间<以毫秒为单位>:
        最短 = 0ms, 最长 = 1ms, 平均 = 0ms
  
```

在 PC1 上打开程序，程序将会自动获取本机网卡设备，将列表打印在控制台上并同时写入日志文件，日志内容如下：

```

1. \Device\NPF_{F1C7BADA-07E4-4157-9C66-C10840F43C6E} (Realtek PCI GBE Family
Controller)

   IP Address: 123.123.110.110
  
```

选择此设备，并设置目的 IP 为 PC2 的 IP: 123.123.110.101，程序将会先行构建一个外来 arp 从而获取当前网卡的 MAC 地址，并将相关信息输出。

```

Enter the interface number (1-1):1
Input the IP Address of destination: 123.123.110.101.

The MAC Address of Adapter 1: 08-57-00-d7-e6-ed
The IP Address of Adapter 1: 123.123.110.110
The IP Address of destination: 123.123.110.101
  
```

Wireshark 抓取的 arp 包如下：

18 22.271662	12:12:12:12:12:12	Broadcast	ARP	42 Who has 123.123.110.110? Tell 18.52.86.120
19 22.271672	Tp-LinkT_d7:e6:ed	12:12:12:12:12:12	ARP	42 123.123.110.110 is at 08:57:00:d7:e6:ed

其中构建的外来虚拟 ARP 的源 IP 为 18.52.86.120，源 MAC 为 12:12:12:12:12:12, 可以看到，通过此方法获取到的本机 MAC 地址为 08-57-00-

d7-e6-ed。

之后，程序则会发送 arp 请求广播，并开始不断抓包进行解析，在收到 PC2 发送的回复 arp 后，通过比对发现源 ip 为 123.123.110.101，停止抓包，并将收到的 PC2 的 MAC 地址打印。日志文件内容如下：

```
Send packet successfully

Catching packets...

(15:16:37) message 1:
request message.
ARP packet length: 42
source mac: 08-57-00-d7-e6-ed
source ip: 123.123.110.110
destination mac: ff-ff-ff-ff-ff-ff
destination ip: 123.123.110.101

(15:16:37) message 2:
reply message.
ARP packet length: 60
source mac: 08-57-00-d7-d0-b9
source ip: 123.123.110.101
destination mac: 08-57-00-d7-e6-ed
destination ip: 123.123.110.110

Get the MAC address of destination: 08-57-00-d7-d0-b9

End of catching...
```

可以看到，程序共抓取到了两条 arp 报文，第一条为 PC1 向局域网内发送的广播报文，第二条为 PC2 向 PC1 发送的回送报文。Wireshark 抓取到的内容如下：

25 30.075273	Tp-LinkT_d7:e6:ed	Broadcast	ARP	42 Who has 123.123.110.101? Tell 123.123.110.110
26 30.075826	Tp-LinkT_d7:d0:b9	Tp-LinkT_d7:e6:ed	ARP	60 123.123.110.101 is at 08:57:00:d7:d0:b9

其中第一条为 PC1 发送的长度为 42 的广播报文，询问 123.123.110.101 的

MAC 地址，第二条为 PC2 发送给 PC1 的报文，告知对应的 MAC 地址为 08-57-00-d7-d0-b9，结果正确。

接下来验证接收功能，在 PC2 上打开接收程序，在 PC1 上利用控制台 ping PC2，程序将会自动抓取此操作相关的 arp 协议。最终生成的日志内容如下：

```
(15:17:50) message 1:
request message.
ARP packet length: 60
source mac: 08-57-00-d7-e6-ed
source ip: 123.123.110.110
destination mac: ff-ff-ff-ff-ff-ff
destination ip: 123.123.110.101

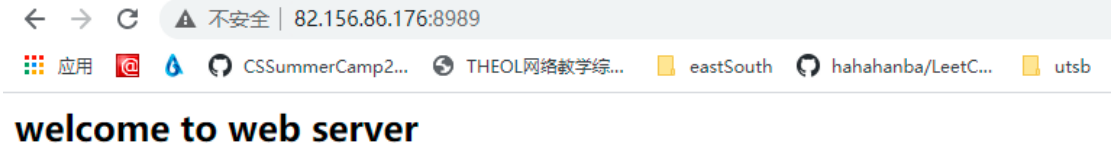
(15:17:50) message 2:
reply message.
ARP packet length: 42
source mac: 08-57-00-d7-d0-b9
source ip: 123.123.110.101
destination mac: 08-57-00-d7-e6-ed
destination ip: 123.123.110.110
```

其中第 1 条为广播报文，第二条为 PC2 发给 PC1 的回送报文。Wireshark 抓取到的内容如下，结果正确：

267	352.670205	Tp-LinkT_d7:e6:ed	Broadcast	ARP	60 Who has 123.123.110.101? Tell 123.123.110.110
268	352.670217	Tp-LinkT_d7:d0:b9	Tp-LinkT_d7:e6:ed	ARP	42 123.123.110.101 is at 08:57:00:d7:d0:b9

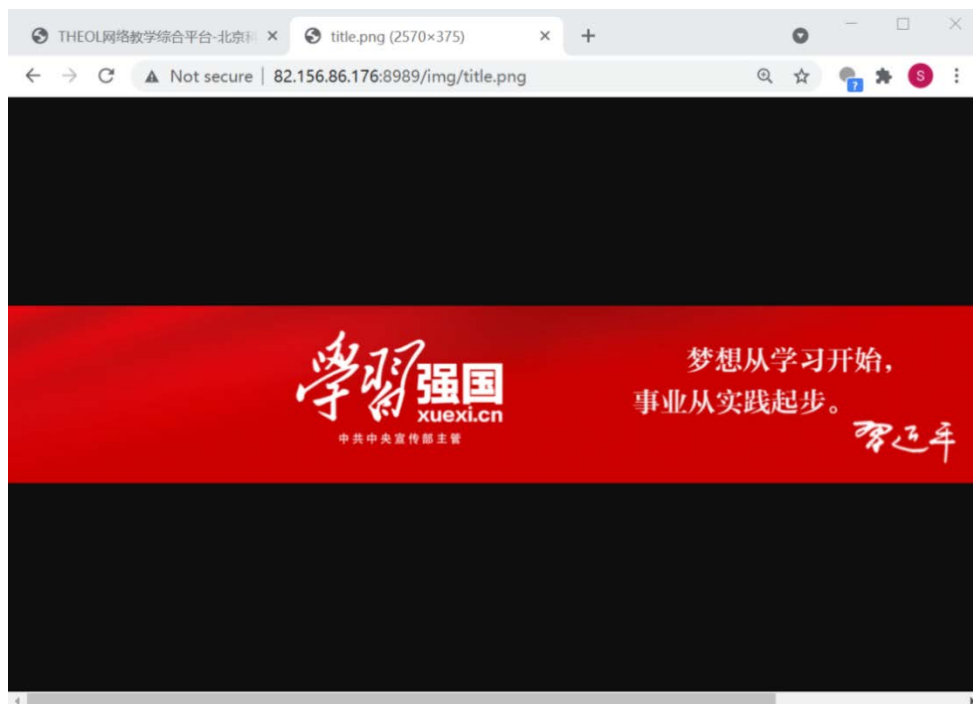
2) 实验二

对于实验二：我们小组同时使用多台设备，通过不同的浏览器（chrome、Microsoft Edge、safari）对网页进行访问，均能正常访问





对于图片的访问，也能正常显示



同时，我们对所实现的 Web 服务器进行了并发测试，访问的是 index 和 404.html，结果如下：

```
root@VM-0-6-ubuntu:/home/ubuntu/http_load# ./http_load -r 1000 -s 5 test.txt
381 fetches, 15 max parallel, 41083 bytes, in 5.02051 seconds
107.829 mean bytes/connection
75.8887 fetches/sec, 8183.03 bytes/sec
msecs/connect: 46.4384 mean, 71.777 max, 43.332 min
msecs/first-response: 82.3795 mean, 131.829 max, 25.396 min
HTTP response codes:
  code 200 -- 184
  code 404 -- 197
root@VM-0-6-ubuntu:/home/ubuntu/http_load# ./http_load -p 1000 -s 5 test.txt
2129 fetches, 1000 max parallel, 229907 bytes, in 5.00008 seconds
107.988 mean bytes/connection
425.793 fetches/sec, 45980.7 bytes/sec
msecs/connect: 242.079 mean, 4088.52 max, 23.189 min
msecs/first-response: 438.421 mean, 4371.38 max, 26.552 min
HTTP response codes:
  code 200 -- 1062
  code 404 -- 1067
root@VM-0-6-ubuntu:/home/ubuntu/http_load#
```

在上面的测试中一共运行了 2129 个请求，最大的并发进程数是 1000，总计传输的数据是 229907bytes，运行时间为 5.00008s，每一个连接平均传输的数据量为 107.988；每秒的响应请求为 425.793msecs，每秒传递的数据为 45980.7bytes/sec，每连接的平均响应时间是 242.079 msecs，最大的响应时间 4088.52 msecs，最小的响应时间 23.189 msecs

总结与讨论：

（本次课程的学习体会收获、设计中存在的问题及可能的改进方向）

通过本学期的计算机网络课程设计，我们对上学期所学的内容有了更深刻的体会。

在实验一中，我们掌握了数据分组的发送和解析，使用 winpcap 实现了封装与发送 ARP 的数据分组，对网络协议的理解有了更直观的认识，掌握了 ARP 的协议数据结构和工作原理及其对协议栈的贡献。

实验二中，我们实现了一个基于 epoll 的 web 服务器，可以并行服务处理多个请求，实现了 HTTP1.0 的功能。在自主学习过程中，我们明白了 HTTP 协议的作用原理，同时根据 HTTP 协议的作用原理，进行编程实现。通过这个实验，我们对计算机网络底层有了新的认识，我们成功使用 epoll 实现异步 web 服务，可以指定线程的数量，同时每一个线程独立工作以充分利用硬件的 cpu 及内存等资源。但是，我们的程序设计还存在着一些问题，如无法访问视频等问题，这是我们的努力改进方向，我们将会继续完善，改善不足。

综上，计算机网络课程设计对于我们理解计算机网络有着莫大的帮助，在

本次实验中，我们接触到了课堂上所学不到的东西，收获颇多。

指导教师评语：[\(空白，半页 A4\)](#)