**华中科技大学计算机学院**

**《计算机通信与网络》实验报告**

实验名称 利用Java(C++)开发网络应用程序

|  |  |  |  |
| --- | --- | --- | --- |
| 姓 名 | 班 级 | 学 号 | 得 分 |
| 潘翔 | IOT1601 | U201614898 |  |

教师评语：

# Lab8 利用Java(C++)开发网络应用程序

## 8.1 环境

### 8.1.1 本地环境

操作系统: Manjaro-4.18.10-1 x86\_64 (Arch-Based Distribution)

开发语言: C/C++

编辑器: Visual Studio Code 1.27.2 x64

编译器: g++ (GCC) 8.2.1 20180831

调试器: GNU gdb (GDB) 8.2

构建工具: GNU Make 4.2.1

网络环境：

Link encap:Ethernet  HWaddr a0:8c:fd:24:5d:4c     
inet addr:222.20.100.153  Bcast:222.20.101.255  Mask:255.255.254.0   
inet6 addr: fe80::2476:27:cd9d:d75b/64 Scope:Link   
inet6 addr: 2001:250:4000:803c:e3c1:b69:d9f2:67b0/64 Scope:Global

8.1.2 服务器环境

操作系统: Ubuntu 18.04 bionic x86\_64 (Linux 4.15.0-20-generic)

开发语言: C/C++

编辑器: Vim

编译器: g++ (GCC) 7.3.0

网络环境：

eth0:

inet 23.105.204.80 netmask 255.255.248.0 broadcast 23.105.207.255

inet6 fe80::a8aa:ff:fe0e:10bc prefixlen 64 scopeid 0x20<link>

ether aa:aa:00:0e:10:bc txqueuelen 1000 (Ethernet)

## 8.2 实验目的

1. 基本掌握利用C++开发环境调试应用程序的方法。
2. 理解基于套接字开发网络应用程序的过程，深入理解Ping工作原理。
3. 深入理解HTTP协议的格式和工作过程，理解Web代理服务器工作原理。

## 8.3 实验内容及步骤

### 8.3.1 myping

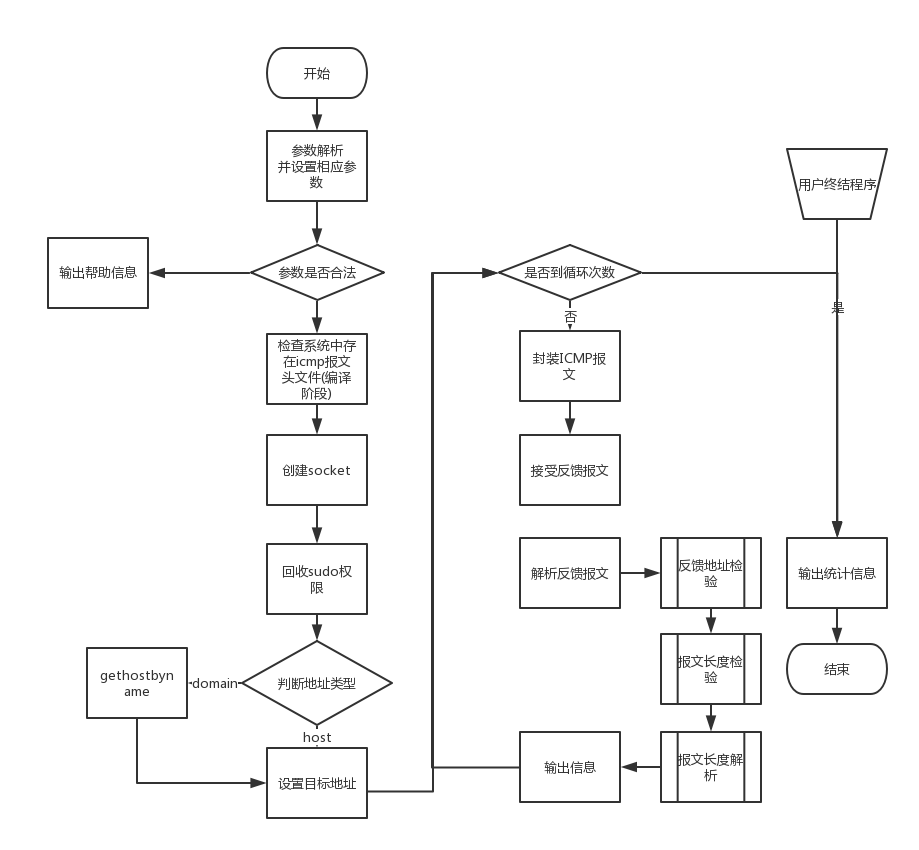


图8-1 myping函数流程图

### 8.3.2 myWebServer

1. server\_base
   1. request 结构体定义
   2. 初始化
   3. 装载资源
   4. 默认的request处理函数
2. server\_http
   1. 初始化服务器
      1. 指定端口
      2. 线程数量
   2. 重写accept函数
      1. m\_io\_service资源池中取出socket
      2. 异步接受
         1. 调用accept函数
         2. 如果错误，调用默认process\_request函数
3. server\_https
   1. 初始化服务器
      1. 指定端口
      2. 线程数量
      3. 证书文件
      4. 私钥文件
   2. 重写accept函数
      1. m\_io\_service资源池中取出socket
      2. 异步接受
         1. 利用SSL对于报文IO Stream加密
         2. 如果错误，调用默认process\_request函数

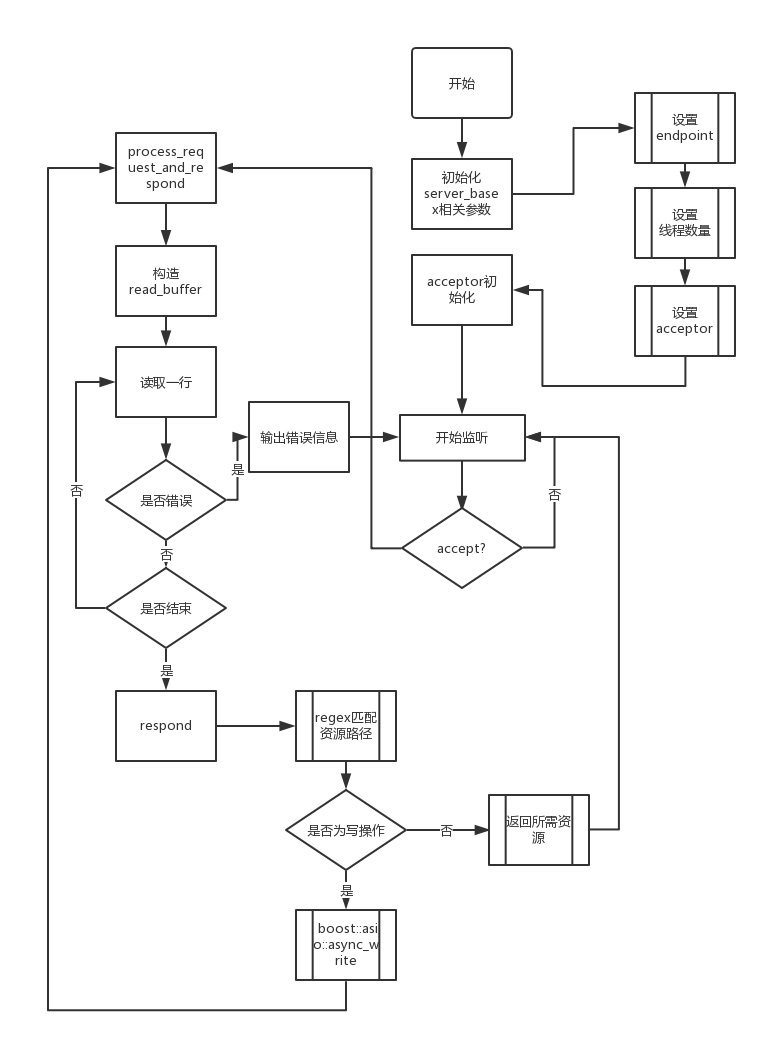


图8-2 server\_base流程图

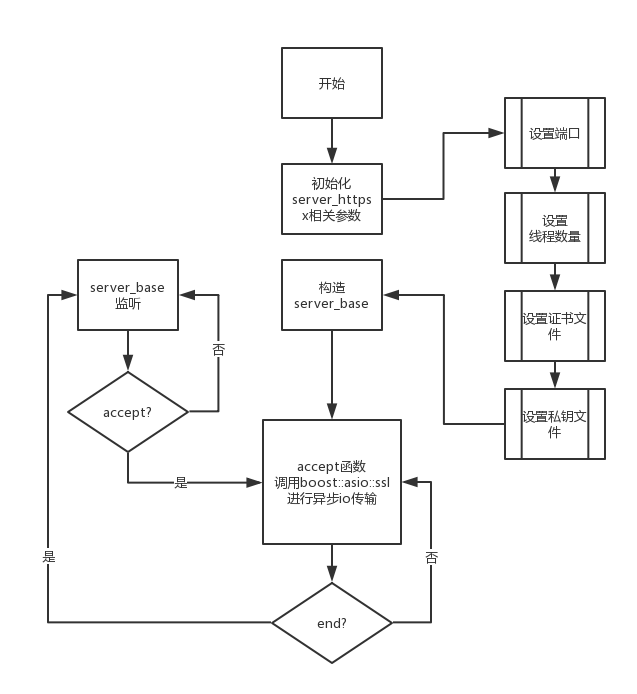


图8-3 server\_https流程图

1. 相关函数接口的说明
   1. HTTP

网络的相关概念已经被抽象为 I/O，只需要关心从这个 I/O 流中获取消 息

* + 1. io\_service来构造boost::asio::ip::tcp::socket 对象作为Socket进行网络I/O传 输
    2. boost::asio::ip::tcp::endpoint指定服务器IP,和地址
    3. boost::asio::ip::tcp::acceptor acceptor(io, endpoint);用于建立连接
    4. 重载accept函数进行I/O数据的处理
  1. HTTPS

对于普通的HTTP流依赖OpenSSL进行加密

* + 1. 更改流为boost::asio::ssl::stream

typedef boost::asio::ssl::stream<boost::asio::ip::tcp::socket> HTTPS;

* + 1. 构建content对象

boost::asio::ssl::context::sslv23

* + 1. 指定证书类型

use\_certificate\_chain\_file()

use\_private\_key\_file()

* 1. I/O操作处理

网络为流式传输

* + 1. 定义流缓冲，用于逐行读取 socket 中的数据

boost::asio::streambuf read\_buffer;

* + 1. 读取使用分隔符的协议

很多网络协议其实都是基于行实现的，协议元素是由 \r\n 符号进行界定，HTTP也是，故在 Boost Asio 中，读取使用async\_read\_untile()分隔符的协议

boost::asio::async\_read\_until(socket, readbuffer, "\r\n\r\n", read\_handler);

* + 1. 判断读取是否成功

boost::system::error\_code

* + 1. 进行下一步操作

boost::asio::async\_read

boost::asio::async\_write

### 8.3.3 myProxyServer

1. 解析配置参数
   1. 设置线程数
   2. 设置端口
   3. 设置IP
2. 构建线程池
3. 构造server
   1. io\_service来构造 ios\_deque(double-ended queue)对象作为容器进行网络I/O传输
   2. boost::asio::ip::tcp::endpoint指定服务器IP,和地址
   3. boost::asio::ip::tcp::acceptor acceptor(io, endpoint);用于建立连接
4. start\_accept开始监听

如有连接请求

* 1. 从线程池取出线程
  2. 创建proxy\_conn new\_connection
  3. 绑定请求处理函数handle\_accept
  4. 绑定错误返回码

1. handle\_accept处理请求

是否错误

* 1. 如果错误，返回错误码
  2. 正确，调用proxy\_conn类进行处理

1. proxy\_conn处理请求
   1. 初始化读取缓冲区
   2. 读取headers
   3. 分析headers
      1. 读取headers构造headersMap
      2. 利用headersMap获得Content-Length

headersMap::iterator it=respHeaders.find("Content-Length");

* + 1. 解析URL
  1. 构建转发headers
  2. 建立conn
  3. 写asio

根据HTTP报文构造转发报文

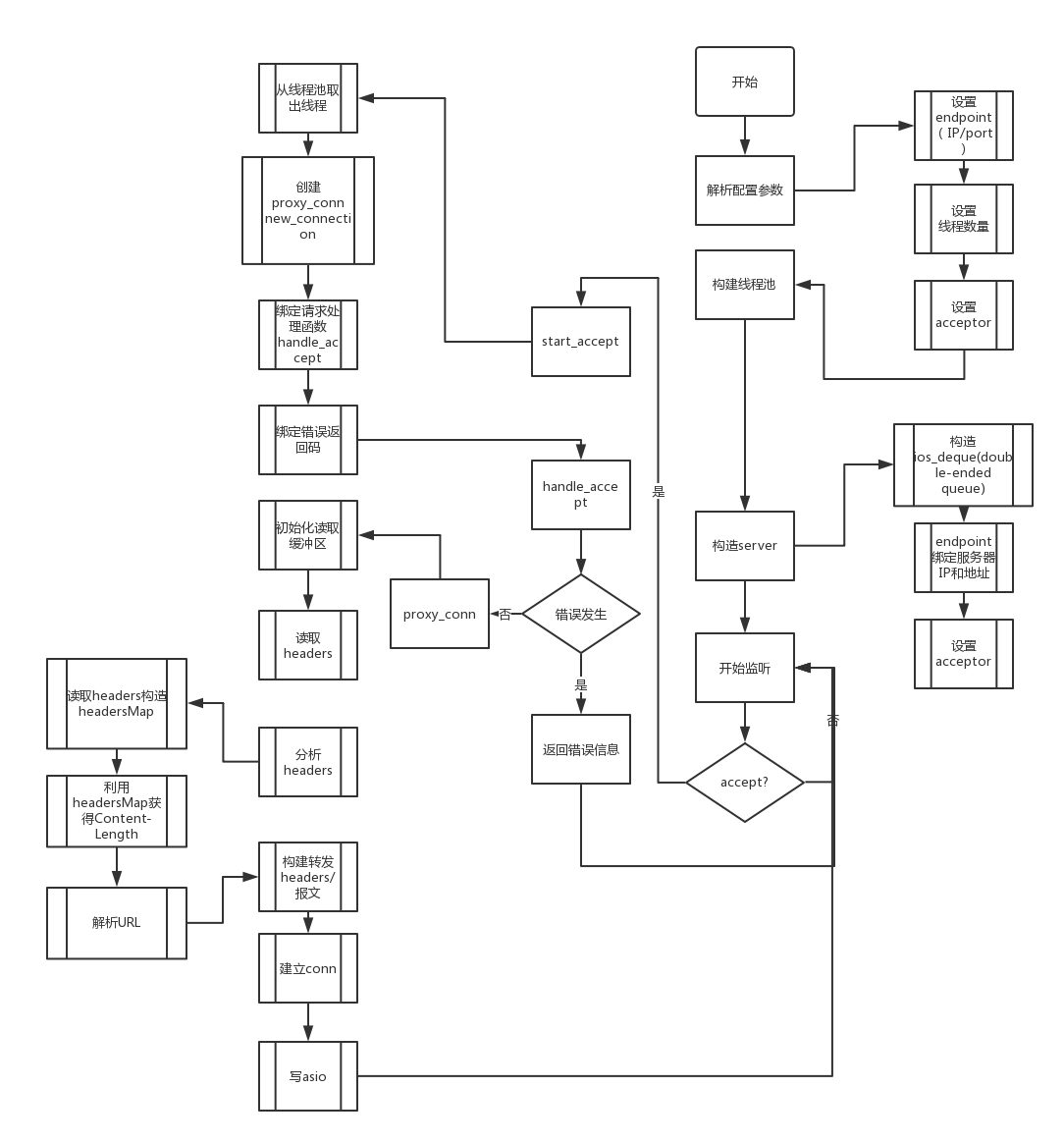


图8-4 my\_proxy\_server流程图

## 8.4 实验结果

### 8.4.1 ping实验结果

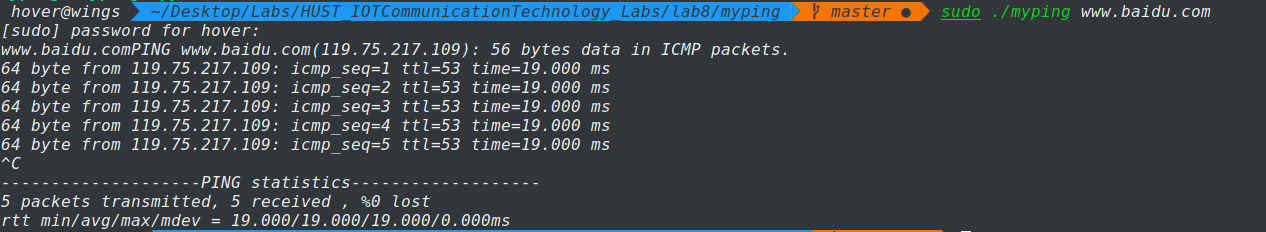


图8-5 myping [www.baidu.com](http://www.baidu.com)

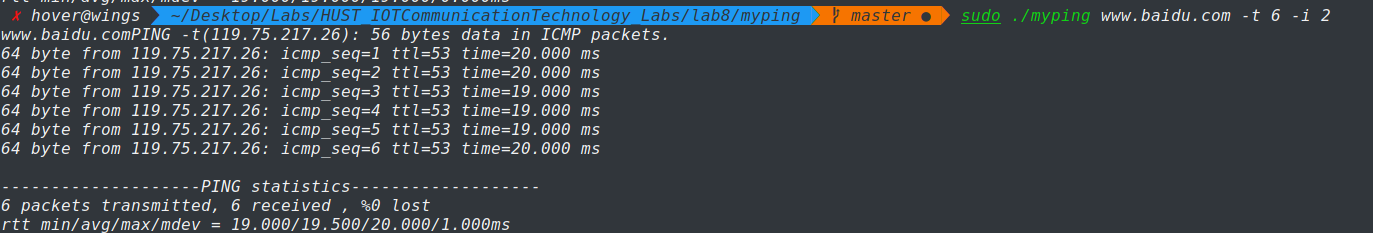


图8-6 myping [www.baidu.com](http://www.baidu.com) 参数解析测试

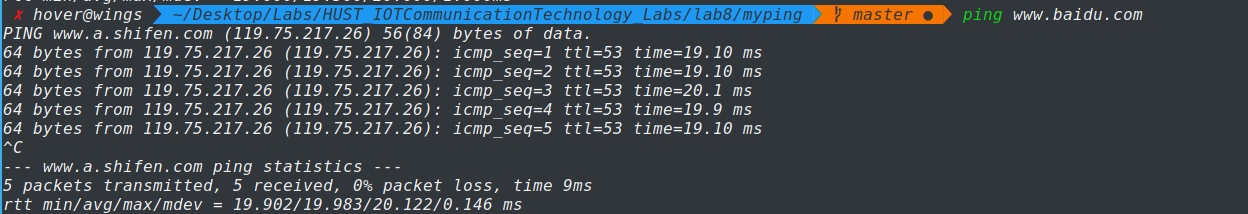


图8-7 ping [www.baidu.com](http://www.baidu.com) 对比图

### 8.4.2 myWebServer

1. 产生私钥

openssl 工具包提供了一个生成 RSA 私钥和 CSR(Certificate Signing Request) 文件的工具。可使用其生成自签名的证书，从而用于供给 HTTPS 服务器使用。

利用openssl产生1024位秘钥，并使用三重 DES 加密方式，并按照 PEM 格式存储：

openssl genrsa -des3 -out server.key 1024

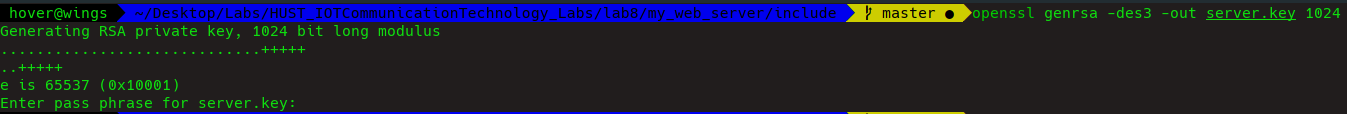


图8-8 生成私钥图

输入hover作为密码

1. 生成CSR

openssl req -new -key server.key -out server.csr

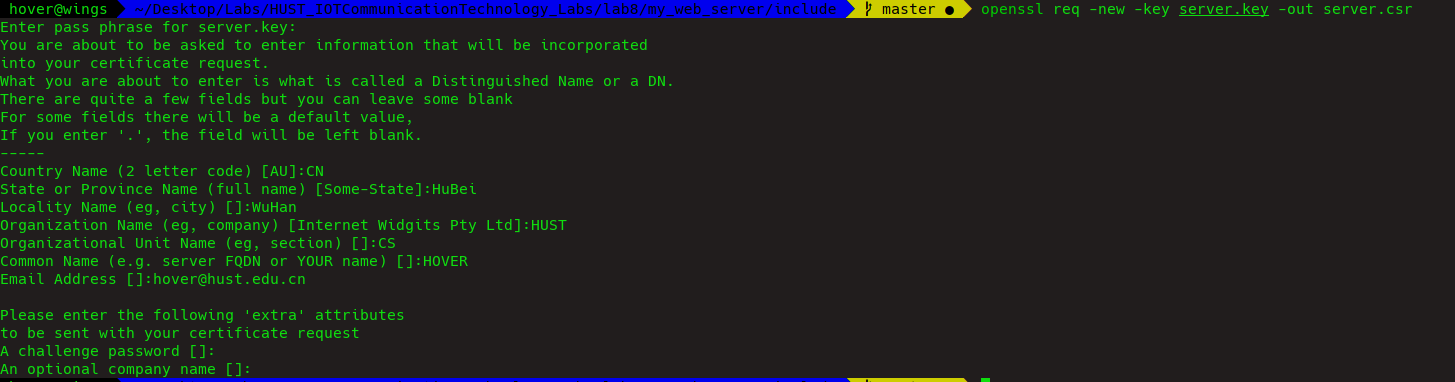


图8-9 生成CSR图

1. 生成自签名证书

openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

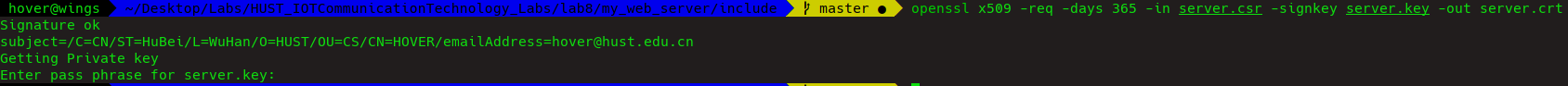


图8-10 生成自签名证书图

1. 运行https服务器(localhost测试)

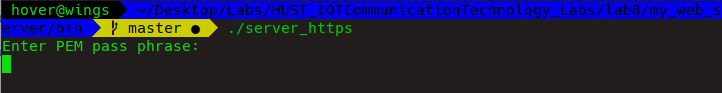


图8-11 https服务器启动

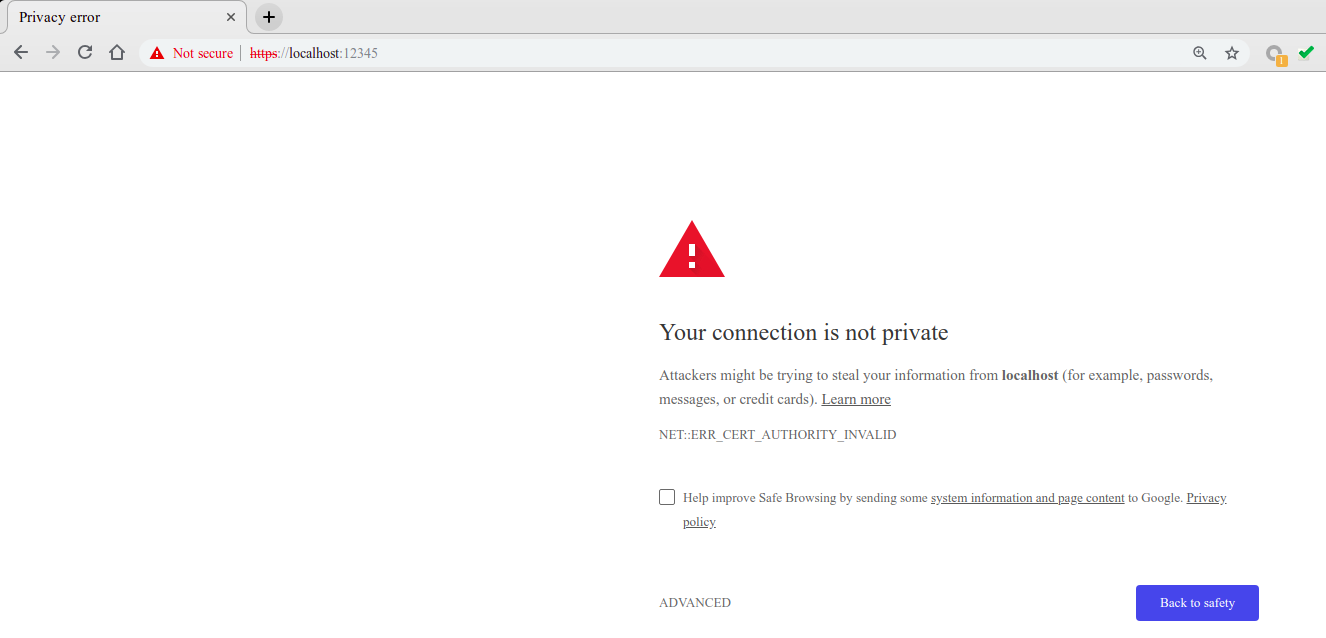


图8-12 浏览器访问拦截图

这是由于证书是自签名的产生的原因。一般情况下，自签名的 SSL 证书可以随意的签发，没有第三方监督的审核，并不能收到浏览器的信任。这就非常容易造成伪造证书的中间人攻击，从而导致劫持 SSL 加密流量。

在创建证书的时候，指定了这个证书会被用于CN域名，而实际上在访问时，访问的URL是localhost，这时浏览器识别到这个不同，也就阻止了这次连接。

使用localhost(unsafe)选项

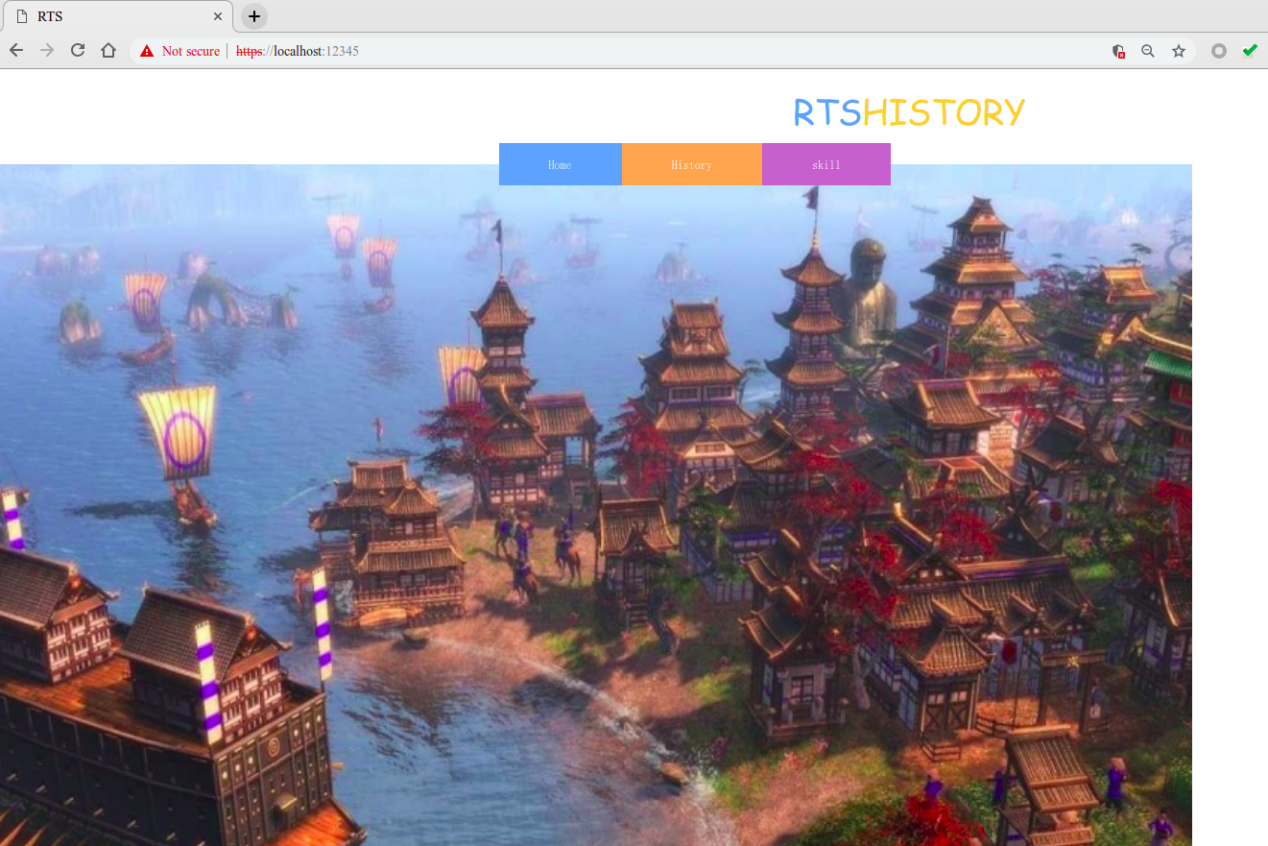


图8-13 浏览器访问图(localhost)

可以看到我们的网页

1. 运行https服务器(服务器测试)



图8-14 接口测试图(23.105.24.80:/12345)

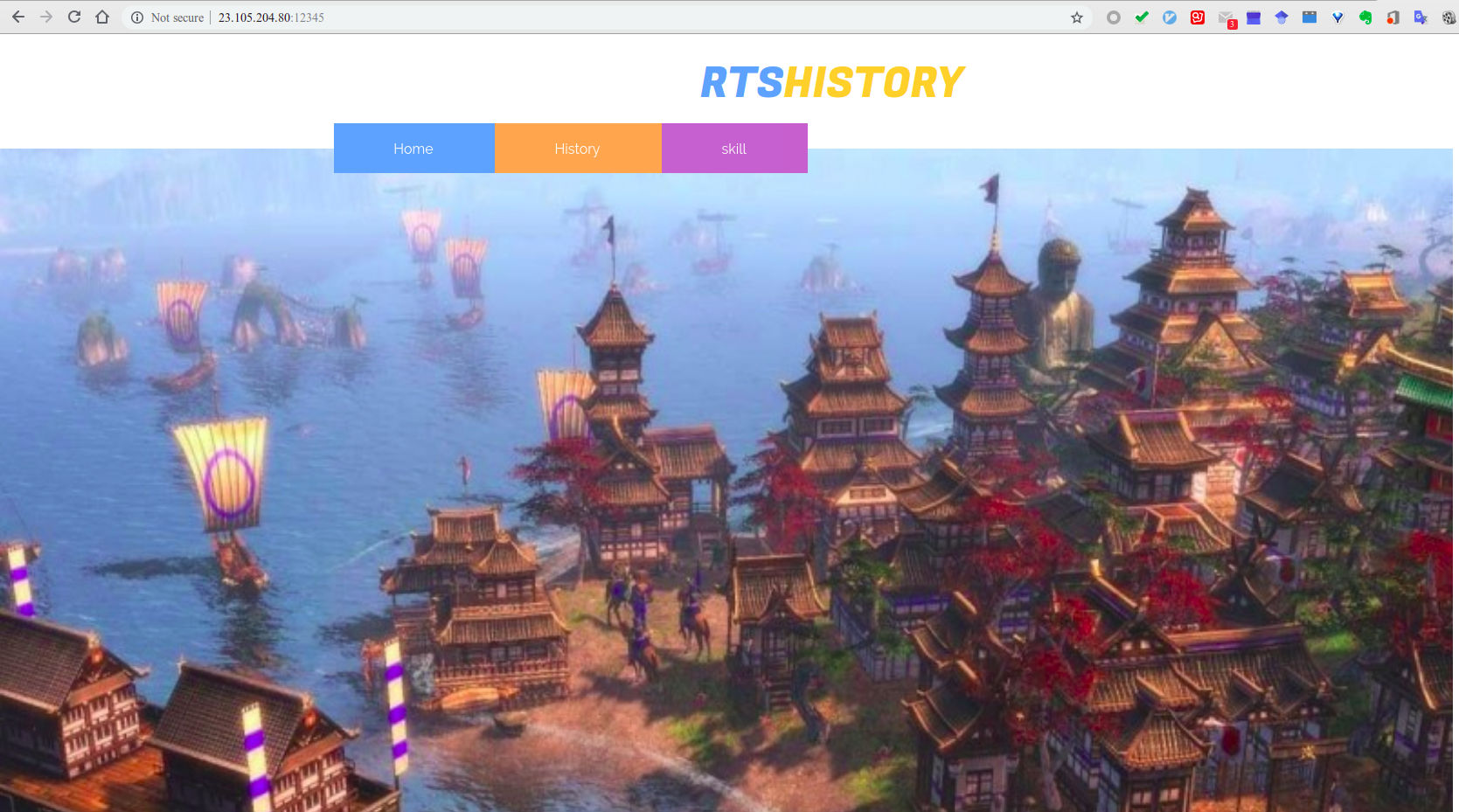


图8-15 浏览器访问图(23.105.24.80:12345)

将web\_server部署到远端服务器

### 8.4.3 myWebProxy

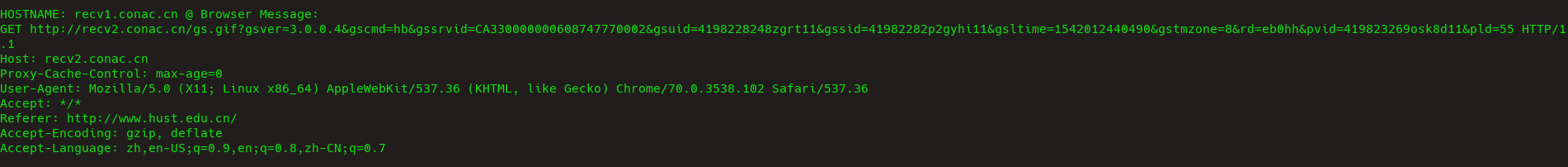


图8-16 proxy本地log图

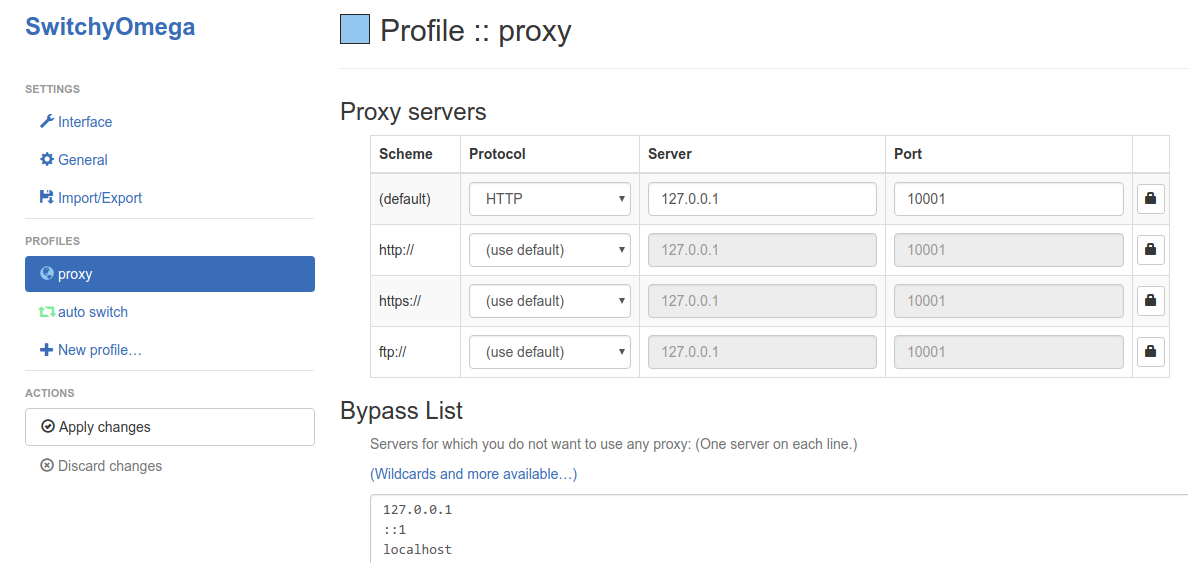


图8-17 浏览器插件设置图

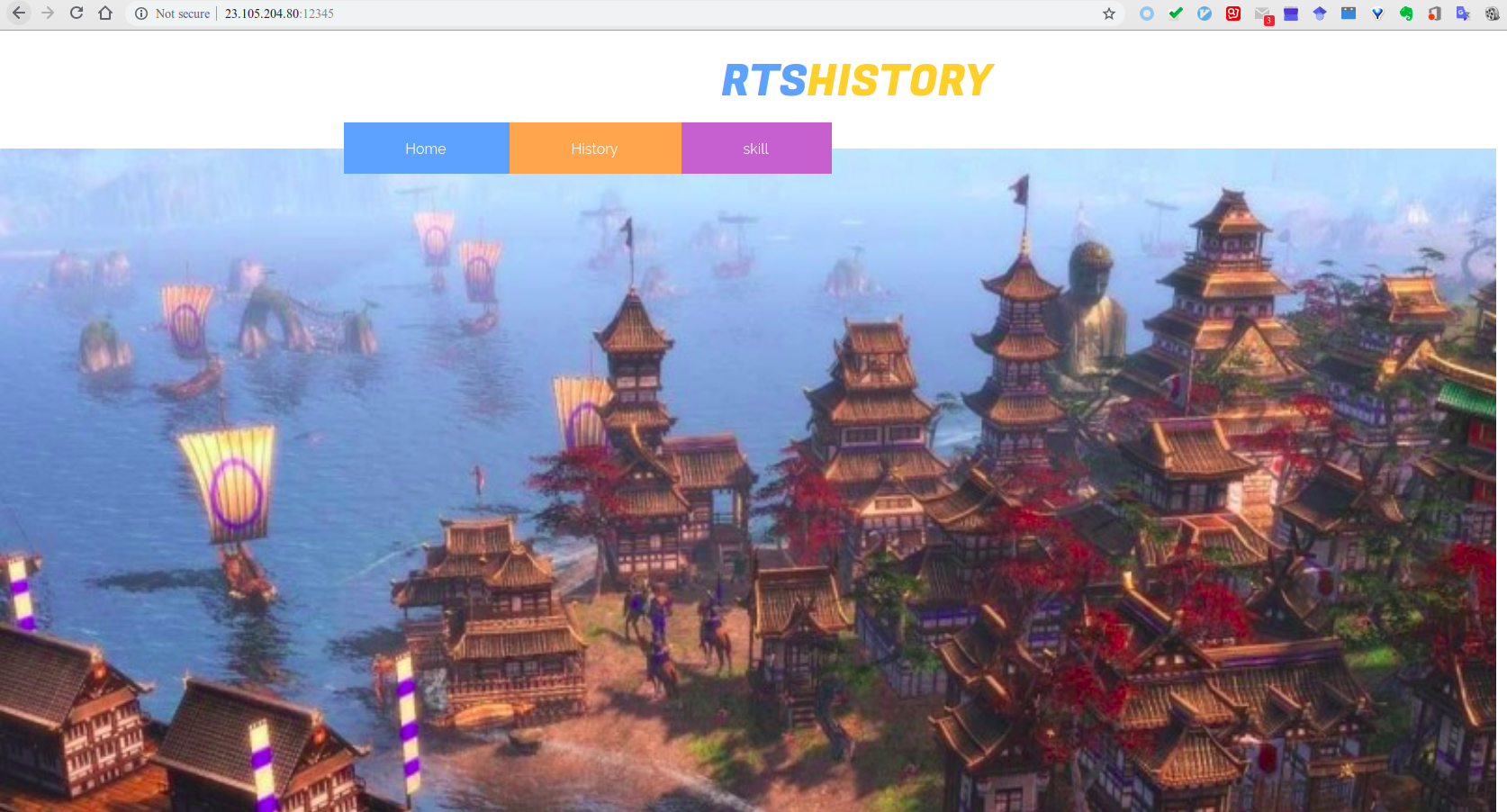


图8-18 代理浏览图

**问题：**

1. 持久连接与非持久连接

HTTP既可以使用非持久连接(nonpersistent connection)，也可以使用持久连接(persistent connection)。HTTP/1.0使用非持久连接，HTTP/1.1默认使用持久连接。

在本次实验中使用HTTP 1.0 Version

1. 程序处理的最长字节

根据Content-Length长度进行判断，同时不能超过buffer size

## 8.5 实验中的问题及心得

在ping的开发过程中，熟悉了UDP和ICMP的报文结构，其中对于大小端的处理，操作系统采用了宏判断的形式，而ICMP报文结构体内使用union形式，对是否回应做出了定义，结构清晰，且进行封装的时候不容易出错，这些值得我们自己在实现协议是进行学习，而在ping的系统实现中，和traceroute一样，可以选择指定协议类型，避免如同lab1中部分服务器出现的对于icmp报文不响应的状态。

Socket的创建需要root权限，但是系统的ping并不需要，是因为使用raw socket的使用，可以采用为应用程序授权的方式免root执行。

WebServer实现过程中，主要依赖Boost库进行，因为其有良好的多线程管理和异步IO操作，同时利用C++11特性，利用智能指针进行服务器资源管理和线程池实现，避免了引用计数错误和内存泄露问题。其中设计server\_base作为基类，定义公共属性，便于不同的服务器进行版本实现，利用server\_http和server\_https进行继承实现主要是其中构造函数和accept方法的重写。使用handler类进行方法类的抽离，减少耦合性，便于添加新的方法。

ProxyServer实现过程中，与webserver类似，既然要提供服务，那么要做到尽量对客户端透明，建立连接的过程中，实际上为转发报文，达到代理效果，本质上，在客户机需要一次代理程序将代理发给代理服务器，在代理服务器再进行一次转换，两次转换完成了最后的功能。ProxyServer能够实现一些比较有用的功能，可以充当软件的网关，进行特定的流量进行控制和筛选，设计上，采用boost::asio，同时接口清晰，便于进一步进行功能的拓展。

此处的代理为较为简单的实现，采用报文转发，即使建立了TCP连接，仍然需要解析报文，效率并不高，新型的代理采用隧道模式，一旦TCP连接建立，则对于TCP两端的报文直接进行盲转发，此时只有一条connection而不是两条。

对于所有需要代理的内容封装于HTTP报文内部，此时可以代理任意的TCP连接，对于采用SSL的HTTPS或者TLS，由于代理没有相应的证书文件，故不能采用普通的代理，此时可以采用tunnel。

编写过程中，因为SwitchyOmega有ByPass且本地不易抓包，故将webServer部署在远端服务器，采用WireShark进行抓包分析。

其中，Boost库为我们提供了一些很好的示范，包括在进行网络流传输的时候，接受到的流可能大于bufferSize,传统的做法为舍弃溢出部分并报错，而推荐做法为解析头文件后对于溢出部分进行拼接。

整个过程熟悉了http和https应用层协议，锻炼了网络编程技巧，并学习了相关协议的实现。

## 参考文献

1. Boost.Asio Docement

<https://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html>

1. UNIX Network Programming,3th
2. C++ IO

<https://en.cppreference.com/w/cpp/io>

# Lab8 附录

## myping

### myping.cpp

/\* FileName: myping.cpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: the Hover's impletation of ping

\* Attention: you may need sudo to run this code

\*/

#include <cstdio>

#include <iostream>

#include <algorithm>

#include <iomanip>

#include <vector>

#include <getopt.h>

#include <stdlib.h>

#include <string.h>

#include <signal.h>

#include <sys/time.h>

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <unistd.h>

#include <netinet/in.h>

#include <netinet/ip.h>

#include <netinet/ip\_icmp.h>

#include <netdb.h>

#include <setjmp.h>

#include <errno.h>

using namespace std;

#define MAX\_WAIT\_TIME 5

#define PACKET\_SIZE 4096

int max\_no\_packets=5;

int interval=1;

char sendpacket[PACKET\_SIZE];

char recvpacket[PACKET\_SIZE];

pid\_t pid;

int sockfd;

int datalen = 56;

int nsend = 0;

int nreceived = 0;

struct sockaddr\_in dest\_addr;

struct sockaddr\_in from\_addr;

struct timeval tvrecv;

vector<double> rtt\_vec;

void statistics(int signo);

unsigned short cal\_chksum(unsigned short \*addr,int len);

int pack(int pack\_no);

void send\_packet(void);

void recv\_packet(void);

int unpack(char \*buf,int len);

void timediff(struct timeval \*out,struct timeval \*in);

bool opt\_t = false; // set ttl

bool opt\_i = false; // interval

void Stop(int signo)

{

statistics(signo);

\_exit(0);

}

int main(int argc,char \*argv[])

{

signal(SIGINT, Stop); //set exit function

char opt;

int option\_index = 0;

static struct option long\_options[] =

{

{"help", no\_argument, NULL, 'h'}

};

char str[256];

strcpy(str,argv[1]);

while ((opt = getopt\_long(argc, argv, "t:i:h", long\_options, &option\_index)) != -1)

{

//printf("%c",opt);

//cout<<argv[optind - 1];

switch (opt)

{

case 't':

max\_no\_packets=atoi(argv[optind - 1]);

opt\_t = true;

break;

case 'i':

interval==atoi(argv[optind - 1]);

opt\_i = true;

break;

case 'h':

opt\_i = true;

break;

}

}

struct hostent \*host; //host entry

struct protoent \*protocol;

unsigned long inaddr = 0l;

int size = 50\*1024; //50k

if(argc < 2)

{

printf("use : %s hostname/IP address \n", argv[0]);

exit(1);

}

if((protocol = getprotobyname("icmLab8p")) == NULL)

{

perror("getprotobyname");

exit(1);

}

// setuid(getpid());

// need root to create socket

if((sockfd = socket(AF\_INET, SOCK\_RAW, protocol->p\_proto)) < 0){

perror("socket error");

exit(1);

}

setuid(getuid()); // recycle root privilage

// case: broadcast address then there will be a lot of reply

// so the buf need enough size

setsockopt(sockfd, SOL\_SOCKET, SO\_RCVBUF, &size, sizeof(size) );

bzero(&dest\_addr, sizeof(dest\_addr));

dest\_addr.sin\_family = AF\_INET;

// domain or address judge

printf("%s",str);

if((inaddr=inet\_addr(str)) == INADDR\_NONE)

{

if((host = gethostbyname(str)) == NULL)

{

perror("gethostbyname error");

exit(1);

}

memcpy((char\*)&dest\_addr.sin\_addr, host->h\_addr, host->h\_length);

}

else

{

memcpy((char\*)&dest\_addr.sin\_addr, (char\*)&inaddr, sizeof(inaddr));

}

pid = getpid();

printf("PING %s(%s): %d bytes data in ICMP packets.\n",argv[1], inet\_ntoa(dest\_addr.sin\_addr), datalen);

send\_packet();

statistics(SIGALRM);

return 0;

}

void statistics(int signo)

{

printf("\n--------------------PING statistics-------------------\n");

printf("%d packets transmitted, %d received , %%%d lost\n",nsend, nreceived, (nsend-nreceived)/nsend\*100);

printf("rtt min/avg/max/mdev = ");

sort(rtt\_vec.begin(), rtt\_vec.end());

double min=rtt\_vec.front();

double max=rtt\_vec[rtt\_vec.size()-1];

double total;

for(vector<double>::iterator iter=rtt\_vec.begin();iter!=rtt\_vec.end();iter++)

{

// cout << (\*iter) << endl;

total+=\*iter;

}

double avg=total/nsend;

double mdev=max-min;

cout<<fixed<<setprecision(3) <<min<<"/"<<avg<<"/"<<max<<"/"<<mdev<<"ms"<<endl;

close(sockfd);

exit(1);

}

/\*

I: addr: check data buffer

check data len(byte)

\*/

unsigned short cal\_chksum(unsigned short \*addr,int len)

{

int sum=0;

int nleft = len;

unsigned short \*w = addr;

unsigned short answer = 0;

while(nleft > 1)

{

sum += \*w++;

nleft -= 2;

}

//if the ICMP head len is odd, then the final data is high bit and add it

if(nleft == 1)

{

\*(unsigned char \*)(&answer) = \*(unsigned char \*)w;

sum += answer; /\* 这里将 answer 转换成 int 整数 \*/

}

sum = (sum >> 16) + (sum & 0xffff); // add high bit and low bit

sum += (sum >> 16); // add overflow

answer = ~sum; // 16 bit checksum

return answer;

}

/\*

I: icmp struct

sequence

O: packed icmp

\*/

int pack(int pack\_no)

{

int packsize;

struct icmp \*icmp;

struct timeval \*tval;

icmp = (struct icmp\*)sendpacket;

icmp->icmp\_type = ICMP\_ECHO; // type of service

icmp->icmp\_code = 0;

icmp->icmp\_cksum = 0;

icmp->icmp\_seq = pack\_no;

icmp->icmp\_id = pid;

packsize = 8 + datalen; //64=8(head)+56

tval = (struct timeval \*)icmp->icmp\_data; /\* 获得icmp结构中最后的数据部分的指针 \*/

gettimeofday(tval, NULL); /\* 将发送的时间填入icmp结构中最后的数据部分 \*/

icmp->icmp\_cksum = cal\_chksum((unsigned short \*)icmp, packsize);/\*填充发送方的校验和\*/

return packsize;

}

void send\_packet()

{

int packetsize;

while(nsend < max\_no\_packets)

{

nsend++;

packetsize = pack(nsend); // set ICMP message head

if(sendto(sockfd, sendpacket, packetsize, 0,(struct sockaddr \*)&dest\_addr, sizeof(dest\_addr)) < 0)

{

perror("sendto error");

continue;

}

recv\_packet();

sleep((uint)interval);

}

}

void recv\_packet()

{

int n;

extern int errno;

signal(SIGALRM,statistics);

int from\_len = sizeof(from\_addr);

while(nreceived < nsend)

{

alarm(MAX\_WAIT\_TIME);

if((n = recvfrom(sockfd, recvpacket, sizeof(recvpacket), 0,(struct sockaddr \*)&from\_addr, (socklen\_t \*)&from\_len)) < 0)

{

if(errno == EINTR)

{

continue;

}

perror("recvfrom error");

continue;

}

gettimeofday(&tvrecv, NULL); // get receive time

if(unpack(recvpacket, n) == -1)

continue;

nreceived++;

}

}

/\*

I:buf IP buf

len IP message len

addr ICMP dest address

O:error code

\*/

int unpack(char \*buf, int len)

{

int iphdrlen;

struct ip \*ip;

struct icmp \*icmp;

struct timeval \*tvsend;

double rtt;

ip = (struct ip \*)buf;

iphdrlen = ip->ip\_hl << 2; //ip head len

icmp = (struct icmp \*)(buf + iphdrlen); // seek to IP message

len -= iphdrlen;

if(len < 8) // less than ICMP head len

{

printf("ICMP packets\'s length is less than 8\n");

return -1;

}

// check ICMP reply

if((icmp->icmp\_type == ICMP\_ECHOREPLY) && (icmp->icmp\_id == pid))

{

tvsend = (struct timeval \*)icmp->icmp\_data;

timediff(&tvrecv, tvsend);

rtt = tvrecv.tv\_sec \* 1000 + tvrecv.tv\_usec / 1000;

rtt\_vec.push\_back(rtt);

printf("%d byte from %s: icmp\_seq=%u ttl=%d time=%.3f ms\n",

len, // total message len

inet\_ntoa(from\_addr.sin\_addr),

icmp->icmp\_seq,

ip->ip\_ttl,

rtt); //ms rtt

return 0;

}

else

return -1;

}

/\*

I:begin\_time

endtime

O:ms diff

\*/

void timediff(struct timeval \*recv, struct timeval \*send)

{

if((recv->tv\_usec -= send->tv\_usec) < 0)

{

--recv->tv\_sec;

recv->tv\_usec += 1000000;

}

recv->tv\_sec -= send->tv\_sec;

}

## my\_web\_server

### handler.hpp

/\* FileName:handler.hpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:handler for handle different request

\*/

#include "server\_base.hpp"

#include <fstream>

using namespace std;

using namespace MyWeb;

template<typename SERVER\_TYPE>

void start\_server(SERVER\_TYPE &server)

{

//TYPE: POST

//TARGET: /string

//RETURN: TARGET str

server.resource["^/string/?$"]["POST"] = [](ostream& response, Request& request)

{

// get (\*request.content) from istream

stringstream ss;

\*request.content >> ss.rdbuf();

string content=ss.str();

response << "HTTP/1.1 200 OK\r\nContent-Length: " << content.length() << "\r\n\r\n" << content;

};

//TYPE: GET

//TARGET: /info

//RETURN: request info

server.resource["^/info/?$"]["GET"] = [](ostream& response, Request& request)

{

stringstream content\_stream;

content\_stream << "<h1>Request:</h1>";

content\_stream << request.method << " " << request.path << " HTTP/" << request.http\_version << "<br>";

for(auto& header: request.header) {

content\_stream << header.first << ": " << header.second << "<br>";

}

//content\_stream len =content.tellp()

content\_stream.seekp(0, ios::end);

response << "HTTP/1.1 200 OK\r\nContent-Length: " << content\_stream.tellp() << "\r\n\r\n" << content\_stream.rdbuf();

};

//TYPE: GET

//REQUEST: GET /match/[[0-9a-zA-Z]]

//RETURN: [[0-9a-zA-Z]]

server.resource["^/match/([0-9a-zA-Z]+)/?$"]["GET"] = [](ostream& response, Request& request)

{

string number=request.path\_match[1];

response << "HTTP/1.1 200 OK\r\nContent-Length: " << number.length() << "\r\n\r\n" << number;

};

//TYPE: GET

//REQUEST: DEFAULT

//RETURN: web/index.html

server.default\_resource["^/?(.\*)$"]["GET"] = [](ostream& response, Request& request)

{

string filename = "www/";

string path = request.path\_match[1];

// prevent .. to visit other location

size\_t last\_pos = path.rfind(".");

size\_t current\_pos = 0;

size\_t pos;

while((pos=path.find('.', current\_pos)) != string::npos && pos != last\_pos)

{

current\_pos = pos;

path.erase(pos, 1);

last\_pos--;

}

filename += path;

ifstream ifs;

if(filename.find('.') == string::npos)

{

if(filename[filename.length()-1]!='/')

filename+='/';

filename += "index.html";

}

ifs.open(filename, ifstream::in);

if(ifs)

{

ifs.seekg(0, ios::end);

size\_t length=ifs.tellg();

ifs.seekg(0, ios::beg);

// copy file to response-stream

response << "HTTP/1.1 200 OK\r\nContent-Length: " << length << "\r\n\r\n" << ifs.rdbuf();

ifs.close();

}

else // file not exist

{

string content="Could not open file "+filename;

response << "HTTP/1.1 400 Bad Request\r\nContent-Length: " << content.length() << "\r\n\r\n" << content;

}

};

// server.run

server.start();

}

### server\_base.hpp

/\* FileName:server\_base.hpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:base class server\_base

\*/

#ifndef SERVER\_BASE\_HPP

#define SERVER\_BASE\_HPP

#include <boost/asio.hpp>

#include <regex>

#include <unordered\_map>

#include <thread>

namespace MyWeb

{

struct Request

{

std::string method; // post/get

std::string path;

std::string http\_version;

std::shared\_ptr<std::istream> content; //use shared\_ptr to count reference num

std::unordered\_map<std::string, std::string> header;

std::smatch path\_match; //regular expression for path match

};

// resource\_type abbreviation

typedef std::map<std::string, std::unordered\_map<std::string,std::function<void(std::ostream&, Request&)>>> resource\_type;

// socket\_type: HTTP/HTTPS

template <typename socket\_type>

class ServerBase

{

public:

resource\_type resource;

resource\_type default\_resource;

// Construct Server

ServerBase(unsigned short port, size\_t num\_threads = 1) :

endpoint(boost::asio::ip::tcp::v4(), port),

acceptor(m\_io\_service, endpoint), //init end point

num\_threads(num\_threads) {}

void start()

{

for(auto it = resource.begin(); it != resource.end(); it++)

{

all\_resources.push\_back(it);

}

for(auto it = default\_resource.begin(); it != default\_resource.end(); it++)

{

all\_resources.push\_back(it);

}

// socket accept for link

accept();

// thread pool

for(size\_t c = 1;c < num\_threads; c++)

{

threads.emplace\_back([this]()

{

m\_io\_service.run();//(num\_threads-1)

});

}

// main thread

m\_io\_service.run();

// waiting for other thread

for(auto& t: threads)

t.join();

}

protected:

// acceptor init dependence

// IO obj need io\_service init

boost::asio::io\_service m\_io\_service;

boost::asio::ip::tcp::endpoint endpoint;

boost::asio::ip::tcp::acceptor acceptor;

// server thread

size\_t num\_threads;

std::vector<std::thread> threads;

std::vector<resource\_type::iterator> all\_resources;

// different implemantation in different server version

virtual void accept() {}

void process\_request\_and\_respond(std::shared\_ptr<socket\_type> socket) const

{

// async\_read\_untile()

//shared\_ptr for tmp onj

auto read\_buffer = std::make\_shared<boost::asio::streambuf>();

boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n\r\n",

[this, socket, read\_buffer](const boost::system::error\_code& ec, size\_t bytes\_transferred)

{

if(!ec)

{

// if out of buffer then add the data out of buffer when finishing analysing the head

size\_t total = read\_buffer->size();

std::istream stream(read\_buffer.get());

auto request = std::make\_shared<Request>();

\*request = parse\_request(stream);

size\_t num\_additional\_bytes = total-bytes\_transferred;

if(request->header.count("Content-Length")>0)

{

boost::asio::async\_read(\*socket, \*read\_buffer,

boost::asio::transfer\_exactly(stoull(request->header["Content-Length"]) - num\_additional\_bytes),

[this, socket, read\_buffer, request](const boost::system::error\_code& ec, size\_t bytes\_transferred) {

if(!ec)

{

request->content = std::shared\_ptr<std::istream>(new std::istream(read\_buffer.get()));

respond(socket, request);

}

});

}

else

{

respond(socket, request);

}

}

});

}

Request parse\_request(std::istream& stream) const

{

Request request;

std::regex e("^([^ ]\*) ([^ ]\*) HTTP/([^ ]\*)$");

std::smatch sub\_match;

std::string line;

getline(stream, line);

line.pop\_back();

// use regex to parse the request

if(std::regex\_match(line, sub\_match, e))

{

request.method = sub\_match[1];

request.path = sub\_match[2];

request.http\_version = sub\_match[3];

bool matched;

e="^([^:]\*): ?(.\*)$";

// other info

do

{

getline(stream, line);

line.pop\_back();

matched=std::regex\_match(line, sub\_match, e);

if(matched)

{

request.header[sub\_match[1]] = sub\_match[2];

}

} while(matched==true);

}

return request;

}

void respond(std::shared\_ptr<socket\_type> socket, std::shared\_ptr<Request> request) const

{

// match path and request for genetrating respond

for(auto res\_it: all\_resources)

{

std::regex e(res\_it->first);

std::smatch sm\_res;

if(std::regex\_match(request->path, sm\_res, e))

{

if(res\_it->second.count(request->method)>0)

{

request->path\_match = move(sm\_res);

auto write\_buffer = std::make\_shared<boost::asio::streambuf>();

std::ostream response(write\_buffer.get());

res\_it->second[request->method](response, \*request);

boost::asio::async\_write(\*socket, \*write\_buffer,

[this, socket, request, write\_buffer](const boost::system::error\_code& ec, size\_t bytes\_transferred)

{

//HTTP Stable(HTTP 1.1):

if(!ec && stof(request->http\_version)>1.05)

process\_request\_and\_respond(socket);

});

return;

}

}

}

}

};

template<typename socket\_type>

class Server : public ServerBase<socket\_type> {};

}

#endif /\* SERVER\_BASE\_HPP \*/

### server\_http.hpp

/\* FileName:server\_http.hpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:server\_http

\*/

#ifndef SERVER\_HTTP\_HPP

#define SERVER\_HTTP\_HPP

#include "server\_base.hpp"

namespace MyWeb

{

typedef boost::asio::ip::tcp::socket HTTP;

template<>

class Server<HTTP> : public ServerBase<HTTP>

{

public:

//with conf file for init

Server(unsigned short port, size\_t num\_threads=1) :

ServerBase<HTTP>::ServerBase(port, num\_threads) {};

private:

// the accept implemantation

void accept()

{

// new socket

auto socket = std::make\_shared<HTTP>(m\_io\_service);

acceptor.async\_accept(\*socket, [this, socket](const boost::system::error\_code& ec)

{

accept();

if(!ec) //error

{

process\_request\_and\_respond(socket);

}

});

}

};

}

#endif /\* SERVER\_HTTP\_HPP \*/

### server\_https.hpp

/\* FileName:server\_https.hpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:server\_https

\*/

#ifndef SERVER\_HTTPS\_HPP

#define SERVER\_HTTPS\_HPP

#include "server\_http.hpp"

#include <boost/asio/ssl.hpp>

namespace MyWeb

{

// HTTPS type

typedef boost::asio::ssl::stream<boost::asio::ip::tcp::socket> HTTPS;

// HTTPS templeate

template<>

class Server<HTTPS> : public ServerBase<HTTPS>

{

public:

// certificate file

// private\_key file

Server(unsigned short port,

size\_t num\_threads,

const std::string& cert\_file,

const std::string& private\_key\_file) :

ServerBase<HTTPS>::ServerBase(port, num\_threads),

context(boost::asio::ssl::context::sslv23)

{

context.use\_certificate\_chain\_file(cert\_file);

// file type

context.use\_private\_key\_file(private\_key\_file, boost::asio::ssl::context::pem);

}

private:

//ssl context

boost::asio::ssl::context context;

// IO stream (ssl context) encryption

void accept()

{

auto socket = std::make\_shared<HTTPS>(m\_io\_service, context);

acceptor.async\_accept

(

(\*socket).lowest\_layer(),

[this, socket](const boost::system::error\_code& ec)

{

accept();

if(!ec)

{

(\*socket).async\_handshake(boost::asio::ssl::stream\_base::server,

[this, socket](const boost::system::error\_code& ec) {

if(!ec) process\_request\_and\_respond(socket);

});

}

});

}

};

}

#endif /\* SERVER\_HTTPS\_HPP \*/

### server\_http\_runner.cpp

/\* FileName:server\_http\_runner.cpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:server\_http\_runner

\*/

#include "server\_http.hpp"

#include "handler.hpp"

using namespace MyWeb;

int main()

{

//port: 12345

//thread num:4

Server<HTTP> server(12345, 4);

start\_server<Server<HTTP>>(server);

return 0;

}

### server\_https\_runner.cpp

/\* FileName:server\_https\_runner.cpp

\* Author:Hover

\* E-Mail:hover@hust.edu.cn

\* GitHub:HoverWings

\* Description:server\_https\_runner

\*/

#include "server\_https.hpp"

#include "handler.hpp"

using namespace MyWeb;

int main()

{

//port: 12345

//thread num:4

//certificate file: server.crt

//private\_key file: server.key

Server<HTTPS> server(12345, 4, "server.crt", "server.key");

start\_server<Server<HTTPS>>(server);

return 0;

}

## my\_proxy\_server

### common.h

/\* FileName: common.h

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: the common include for program

\*/

#ifndef \_COMMON\_H

#define \_COMMON\_H

#include <boost/asio.hpp>

#include <boost/shared\_ptr.hpp>

#include <boost/enable\_shared\_from\_this.hpp>

#include <boost/algorithm/string.hpp>

#include <boost/lexical\_cast.hpp>

#include <boost/regex.hpp>

#include <boost/bind.hpp>

#include <boost/thread/thread.hpp>

#include <iostream>

#include <string>

namespace ba=boost::asio;

namespace bs=boost::system;

typedef boost::shared\_ptr<ba::ip::tcp::socket> socket\_ptr;

typedef boost::shared\_ptr<ba::io\_service> io\_service\_ptr;

#endif /\* \_COMMON\_H \*/

### proxy\_conn.hpp

/\* FileName: proxy\_conn.hpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: process the connection process of proxy server

\*/

#ifndef \_PROXY\_CONN\_H

#define \_PROXY\_CONN\_H 1

#include "common.h"

#include <boost/unordered\_map.hpp>

class connection : public boost::enable\_shared\_from\_this<connection>

{

public:

typedef boost::shared\_ptr<connection> pointer;

static pointer create(ba::io\_service& io\_service) {

return pointer(new connection(io\_service));

}

ba::ip::tcp::socket& socket() {

return bsocket\_;

}

/// Start read data of request from browser

void start();

private:

connection(ba::io\_service& io\_service);

/// Read header of HTTP request from browser

void handle\_browser\_read\_headers(const bs::error\_code& err, size\_t len);

/// Start connecting to the web-server, initially to resolve the DNS-name of Web server into the IP address

void start\_connect();

void handle\_resolve(const boost::system::error\_code& err,

ba::ip::tcp::resolver::iterator endpoint\_iterator);

void handle\_connect(const boost::system::error\_code& err,

ba::ip::tcp::resolver::iterator endpoint\_iterator, const bool first\_time);

/// Write data to the web-server

void start\_write\_to\_server();

void handle\_server\_write(const bs::error\_code& err, size\_t len);

/// Read header of data returned from the web-server

void handle\_server\_read\_headers(const bs::error\_code& err, size\_t len);

/// Reading data from a Web server, and writing it to the browser

void handle\_browser\_write(const bs::error\_code& err, size\_t len);

void handle\_server\_read\_body(const bs::error\_code& err, size\_t len);

/// Close both sockets: for browser and web-server

void shutdown();

ba::io\_service& io\_service\_;

ba::ip::tcp::socket bsocket\_;

ba::ip::tcp::socket ssocket\_;

ba::ip::tcp::resolver resolver\_;

bool proxy\_closed;

bool isPersistent;

int32\_t RespLen;

int32\_t RespReaded;

boost::array<char, 8192> bbuffer;

boost::array<char, 8192> sbuffer;

std::string fURL;

std::string fHeaders;

std::string fNewURL;

std::string fMethod;

std::string fReqVersion;

std::string fServer;

std::string fPort;

bool isOpened;

std::string fReq;

typedef boost::unordered\_map<std::string,std::string> headersMap;

headersMap reqHeaders, respHeaders;

void parseHeaders(const std::string& h, headersMap& hm);

};

#endif /\* \_proxy\_CONN\_H \*/

### proxy\_conn.cpp

/\* FileName: proxy\_conn.cpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: process the connection process of proxy server

\*/

#include "proxy\_conn.hpp"

connection::connection(ba::io\_service& io\_service) : io\_service\_(io\_service),

bsocket\_(io\_service),

ssocket\_(io\_service),

resolver\_(io\_service),

proxy\_closed(false),

isPersistent(false),

isOpened(false)

{

fHeaders.reserve(8192);

}

/\*

D: Start read data of request from browser

\*/

void connection::start()

{

std::cout << "start" << std::endl;

fHeaders.clear();

reqHeaders.clear();

respHeaders.clear();

handle\_browser\_read\_headers(bs::error\_code(), 0);

}

/\*\*

I: err

len

D: Read header of HTTP request from browser

\*/

void connection::handle\_browser\_read\_headers(const bs::error\_code& err, size\_t len)

{

// std::cout << "handle\_browser\_read\_headers. Error: " << err << ", len=" << len << std::endl;

if(!err)

{

if(fHeaders.empty())

{

fHeaders=std::string(bbuffer.data(),len);

}

else

{

fHeaders+=std::string(bbuffer.data(),len);

}

if(fHeaders.find("\r\n\r\n") == std::string::npos) // going to read rest of headers

{

std::cout << fHeaders << std::endl;

async\_read(bsocket\_, ba::buffer(bbuffer), ba::transfer\_at\_least(1),

boost::bind(&connection::handle\_browser\_read\_headers,

shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

}

else // analyze headers

{

//std::cout << "fHeaders:\n" << fHeaders << std::endl;

std::string::size\_type idx=fHeaders.find("\r\n");

std::string reqString=fHeaders.substr(0,idx);

fHeaders.erase(0,idx+2);

idx=reqString.find(" ");

if(idx == std::string::npos)

{

std::cout << "Bad first line: " << reqString << std::endl;

return;

}

fMethod=reqString.substr(0,idx);

reqString=reqString.substr(idx+1);

idx=reqString.find(" ");

if(idx == std::string::npos)

{

std::cout << "Bad first line of request: " << reqString << std::endl;

return;

}

fURL=reqString.substr(0,idx);

fReqVersion=reqString.substr(idx+1);

idx=fReqVersion.find("/");

if(idx == std::string::npos)

{

std::cout << "Bad first line of request: " << reqString << std::endl;

return;

}

fReqVersion=fReqVersion.substr(idx+1);

// string outputs to console completely, even when using multithreading

std::cout << std::string("\n fMethod: " + fMethod + ", fURL: " + fURL + ", fReqVersion: " + fReqVersion + "\n");

// analyze headers, etc

parseHeaders(fHeaders,reqHeaders);

//

start\_connect();

}

} else {

shutdown();

}

}

/\*\*

\* Start connecting to the web-server, initially to resolve the DNS-name of web-server into the IP address

\*

\*/

void connection::start\_connect()

{

std::string server="";

std::string port="80";

boost::regex rHTTP("http://(.\*?)(:(\\d+))?(/.\*)");

boost::smatch m;

// std::cout<<fURL;

if(boost::regex\_search(fURL, m, rHTTP, boost::match\_extra))

{

server=m[1].str();

if(m[2].str() != "")

{

port=m[3].str();

}

fNewURL=m[4].str();

}

if(server.empty())

{

std::cout << "Can't parse URL "<< std::endl;

return;

}

// std::cout << server << " " << port << " " << fNewURL << std::endl;

if(!isOpened || server != fServer || port != fPort) {

fServer=server;

fPort=port;

ba::ip::tcp::resolver::query query(server, port);

resolver\_.async\_resolve(query,

boost::bind(&connection::handle\_resolve, shared\_from\_this(),

boost::asio::placeholders::error,

boost::asio::placeholders::iterator));

}

else

{

start\_write\_to\_server();

}

}

/\*\*

\* If successful, after the resolved DNS-names of web-server into the IP addresses, try to connect

\*

\* @param err

\* @param endpoint\_iterator

\*/

void connection::handle\_resolve(const boost::system::error\_code& err,

ba::ip::tcp::resolver::iterator endpoint\_iterator) {

// std::cout << "handle\_resolve. Error: " << err.message() << "\n";

if (!err) {

const bool first\_time = true;

handle\_connect(boost::system::error\_code(), endpoint\_iterator, first\_time);

}else {

shutdown();

}

}

/\*\*

\* Try to connect to the web-server

\*

\* @param err

\* @param endpoint\_iterator

\*/

void connection::handle\_connect(const boost::system::error\_code& err,

ba::ip::tcp::resolver::iterator endpoint\_iterator, const bool first\_time) {

// std::cout << "handle\_connect. Error: " << err << "\n";

if (!err && !first\_time)

{

isOpened=true;

start\_write\_to\_server();

}

else if

(endpoint\_iterator != ba::ip::tcp::resolver::iterator())

{

//ssocket\_.close();

ba::ip::tcp::endpoint endpoint = \*endpoint\_iterator;

ssocket\_.async\_connect(endpoint,

boost::bind(&connection::handle\_connect, shared\_from\_this(),

boost::asio::placeholders::error,

++endpoint\_iterator, false));

} else {

shutdown();

}

}

/\*\*

\* Write data to the web-server

\*

\*/

void connection::start\_write\_to\_server()

{

fReq=fMethod;

fReq+=" ";

fReq+=fNewURL;

fReq+=" HTTP/";

fReq+="1.0";

// fReq+=fReqVersion;

fReq+="\r\n";

fReq+=fHeaders;

std::cout << "Request: " << fReq << std::endl;

ba::async\_write(ssocket\_, ba::buffer(fReq),

boost::bind(&connection::handle\_server\_write, shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

fHeaders.clear();

}

/\*\*

\* If successful, read the header that came from a web server

\*

\* @param err

\* @param len

\*/

void connection::handle\_server\_write(const bs::error\_code& err, size\_t len) {

std::cout << "handle\_server\_write. Error: " << err << ", len=" << len << std::endl;

if(!err)

{

handle\_server\_read\_headers(bs::error\_code(), 0);

}

else

{

shutdown();

}

}

/\*\*

\* Read header of data returned from the web-server

\*

\* @param err

\* @param len

\*/

void connection::handle\_server\_read\_headers(const bs::error\_code& err, size\_t len)

{

std::cout << "handle\_server\_read\_headers. Error: " << err << ", len=" << len << std::endl;

if(!err)

{

std::string::size\_type idx;

if(fHeaders.empty())

fHeaders=std::string(sbuffer.data(),len);

else

fHeaders+=std::string(sbuffer.data(),len);

idx=fHeaders.find("\r\n\r\n");

if(idx == std::string::npos)

{ // going to read rest of headers

async\_read(ssocket\_, ba::buffer(sbuffer), ba::transfer\_at\_least(1),

boost::bind(&connection::handle\_server\_read\_headers,

shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

}

else

{ // analyze headers

std::cout << "Response: " << fHeaders << std::endl;

RespReaded=len-idx-4;

idx=fHeaders.find("\r\n");

std::string respString=fHeaders.substr(0,idx);

RespLen = -1;

parseHeaders(fHeaders.substr(idx+2),respHeaders);

std::string reqConnString="",respConnString="";

std::string respVersion=respString.substr(respString.find("HTTP/")+5,3);

headersMap::iterator it=respHeaders.find("Content-Length");

if(it != respHeaders.end())

RespLen=boost::lexical\_cast<int>(it->second);

it=respHeaders.find("Connection");

if(it != respHeaders.end())

respConnString=it->second;

it=reqHeaders.find("Connection");

if(it != reqHeaders.end())

reqConnString=it->second;

isPersistent=(

((fReqVersion == "1.1" && reqConnString != "close") ||

(fReqVersion == "1.0" && reqConnString == "keep-alive")) &&

((respVersion == "1.1" && respConnString != "close") ||

(respVersion == "1.0" && respConnString == "keep-alive")) &&

RespLen != -1);

// std::cout << "RespLen: " << RespLen << " RespReaded: " << RespReaded

// << " isPersist: " << isPersistent << std::endl;

// sent data

ba::async\_write(bsocket\_, ba::buffer(fHeaders),

boost::bind(&connection::handle\_browser\_write,

shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

}

}

else

{

shutdown();

}

}

/\*\*

I: err

len

D; Writing data to the browser, are recieved from web-server

\*/

void connection::handle\_browser\_write(const bs::error\_code& err, size\_t len) {

// std::cout << "handle\_browser\_write. Error: " << err << " " << err.message()

// << ", len=" << len << std::endl;

if(!err)

{

if(!proxy\_closed && (RespLen == -1 || RespReaded < RespLen))

async\_read(ssocket\_, ba::buffer(sbuffer,len), ba::transfer\_at\_least(1),

boost::bind(&connection::handle\_server\_read\_body,

shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

else

{

// shutdown();

if(isPersistent && !proxy\_closed)

{

std::cout << "Starting read headers from browser, as connection is persistent" << std::endl;

start();

}

}

}

else

{

shutdown();

}

}

/\*

I: err

len

D: Reading data from a Web server, for the writing them to the browser

\*/

void connection::handle\_server\_read\_body(const bs::error\_code& err, size\_t len)

{

// std::cout << "handle\_server\_read\_body. Error: " << err << " " << err.message()

// << ", len=" << len << std::endl;

if(!err || err == ba::error::eof)

{

RespReaded+=len;

// std::cout << "len=" << len << " resp\_readed=" << RespReaded << " RespLen=" << RespLen<< std::endl;

if(err == ba::error::eof)

proxy\_closed=true;

ba::async\_write(bsocket\_, ba::buffer(sbuffer,len),

boost::bind(&connection::handle\_browser\_write,

shared\_from\_this(),

ba::placeholders::error,

ba::placeholders::bytes\_transferred));

}

else

{

shutdown();

}

}

/\*\*

D: Close both sockets: for browser and web-server

\*/

void connection::shutdown()

{

ssocket\_.close();

bsocket\_.close();

}

void connection::parseHeaders(const std::string& h, headersMap& hm)

{

std::string str(h);

std::string::size\_type idx;

std::string t;

while((idx=str.find("\r\n")) != std::string::npos)

{

t=str.substr(0,idx);

str.erase(0,idx+2);

if(t == "")

break;

idx=t.find(": ");

if(idx == std::string::npos) {

std::cout << "Bad header line: " << t << std::endl;

break;

}

// std::cout << "Name: " << t.substr(0,idx)

// << " Value: " << t.substr(idx+2) << std::endl;

hm.insert(std::make\_pair(t.substr(0,idx),t.substr(idx+2)));

}

}

### proxy\_server.hpp

/\* FileName: proxy\_server.hpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: the class of proxy server

\*/

#ifndef \_PROXY\_SERVER\_H

#define \_PROXY\_SERVER\_H 1

#include "common.h"

#include "proxy\_conn.hpp"

#include <deque>

typedef std::deque<io\_service\_ptr> ios\_deque;

class server

{

public:

server(const ios\_deque& io\_services, int port=10001, std::string interface\_address = "");

private:

void start\_accept();

void handle\_accept(connection::pointer new\_connection, const bs::error\_code& error);

ios\_deque io\_services\_;

const ba::ip::tcp::endpoint endpoint\_; /\*\*< object, that points to the connection endpoint \*/

ba::ip::tcp::acceptor acceptor\_; /\*\*< object, that accepts new connections \*/

};

#endif /\* \_proxy\_SERVER\_H \*/

### proxy\_server.cpp

/\* FileName: proxy\_server.cpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: the class of proxy server

\*/

#include "proxy\_server.hpp"

server::server(const ios\_deque& io\_services, int port, std::string interface\_address)

: io\_services\_(io\_services),

endpoint\_(interface\_address.empty()?

(ba::ip::tcp::endpoint(ba::ip::tcp::v4(), port)): // INADDR\_ANY for v4 (in6addr\_any if the fix to v6)

ba::ip::tcp::endpoint(ba::ip::address().from\_string(interface\_address), port) ), // specified ip address

acceptor\_(\*io\_services.front(), endpoint\_) // By default set option to reuse the address (i.e. SO\_REUSEADDR)

{

std::cout << endpoint\_.address().to\_string() << ":" << endpoint\_.port() << std::endl;

// std::cout << "server::server" << std::endl;

start\_accept();

}

void server::start\_accept()

{

std::cout << "server::start\_accept" << std::endl;

// Round robin.

io\_services\_.push\_back(io\_services\_.front());

io\_services\_.pop\_front();

connection::pointer new\_connection = connection::create(\*io\_services\_.front());

acceptor\_.async\_accept(new\_connection->socket(),

boost::bind(&server::handle\_accept, this, new\_connection,

ba::placeholders::error));

}

void server::handle\_accept(connection::pointer new\_connection, const bs::error\_code& error)

{

std::cout << "server::handle\_accept" << std::endl;

if (!error)

{

new\_connection->start();

start\_accept();

}

std::cout <<error << std::endl;

}

### proxy\_runner.cpp

/\* FileName: proxy\_runner.cpp

\* Author: Hover

\* E-Mail: hover@hust.edu.cn

\* GitHub: HoverWings

\* Description: the main fun of proxy

\*/

#include "proxy\_server.hpp"

int main(int argc, char\*\* argv)

{

try

{

int thread\_num = 2, port = 10001;

std::string interface\_address;

// std::cout<<argv[1];

if(argc > 1)

thread\_num = boost::lexical\_cast<int>(argv[1]);

// read port number from command line, if provided

if(argc > 2)

port = boost::lexical\_cast<int>(argv[2]);

// read local interface address from command line, if provided

if(argc > 3)

interface\_address = argv[3];

ios\_deque io\_services;

std::deque<ba::io\_service::work> io\_service\_work;

boost::thread\_group thr\_grp;

for (int i = 0; i < thread\_num; ++i)

{

io\_service\_ptr ios(new ba::io\_service);

io\_services.push\_back(ios);

io\_service\_work.push\_back(ba::io\_service::work(\*ios));

thr\_grp.create\_thread(boost::bind(&ba::io\_service::run, ios));

}

server server(io\_services, port, interface\_address);

thr\_grp.join\_all();

}

catch (std::exception& e)

{

std::cerr << e.what() << std::endl;

}

return 0;

}