《计算机网络》实验报告 1

HTTP 代理服务器的设计与实现

童超宇

院 （系）：计算机科学与技术学院　　专　　业：计算机科学与技术

学　　号 ：1152130106　　指导教师：刘亚维

2018 年 4 月

目录

1 实验目的

熟悉并掌握 Socket 网络编程的过程与技术;深入理解 HTTP 协议， 掌握 HTTP 代理服务器的基本工作原理;掌握 HTTP 代理服务器设计与编程实现的基本技能。

2 实验内容

设计并实现一个基本 HTTP 代理服务器。要求在指定端口（例如 8080）接收来自客户的 HTTP 请求并且根据其中的 URL 地址访问该地址所指向的 HTTP 服务器（原服务器），接收 HTTP 服务器的响应报文，并将响应报文转发给对应的客户进行浏览。

设计并实现一个支持 Cache 功能的 HTTP 代理服务器。要求能缓存原服务器响应的对象，并能够通过修改请求报文（添加 if-modified-since 头行），向原服务器确认缓存对象是否是最新版本。

扩展 HTTP 代理服务器，支持如下功能:

a) 网站过滤:允许/不允许访问某些网站;

b) 用户过滤:支持/不支持某些用户访问外部网站;

c) 网站引导:将用户对某个网站的访问引导至一个模拟网站（钓鱼）。

3 实验原理

3.1 Socket 编程的客户端和服务器端主要步骤

服务器端

1. 使用 socket 函数创建一个 socket 描述符，来唯一标识一个 socket。函数原型 int socket(int domain, int type, int protocol)
2. 使用 bind 函数绑定 IP 地址，端口信息等。函数原型 int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen)
3. 使用 listen 函数进行监听创建的 socket。函数原型 int listen(int sockfd, int backlog)
4. 使用 accept 函数接收请求，此时 socket 连接也就建立好了。函数原型 int accept(int sockfd, struct sockaddr * addr, socklen_t * addrlen)
5. 使用 read(),write()等函数调用网络 I/O 进行读写操作，来实现网络中不同进程之间的通信
6. 使用 close 函数关闭网络连接，即关系相应的 socket 描述字。函数原型 int

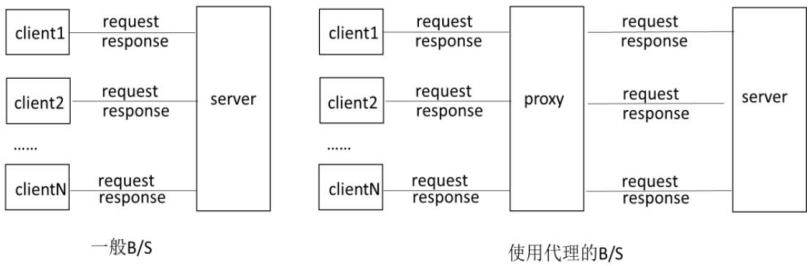
close(int fd)

客户端

1. 创建 socket
2. 绑定 IP 地址，端口信息
3. 设置要连接对方的 IP 地址和端口属性
4. 使用 connect 函数连接服务器。函数原型为 int connect(int sockfd, const struct sockaddr * addr, socklen_t addrlen)
5. 使用 read(),write()等函数进行网络 I/O 的读写
6. 关闭网络连接

3.2 HTTP 代理服务器的基本原理

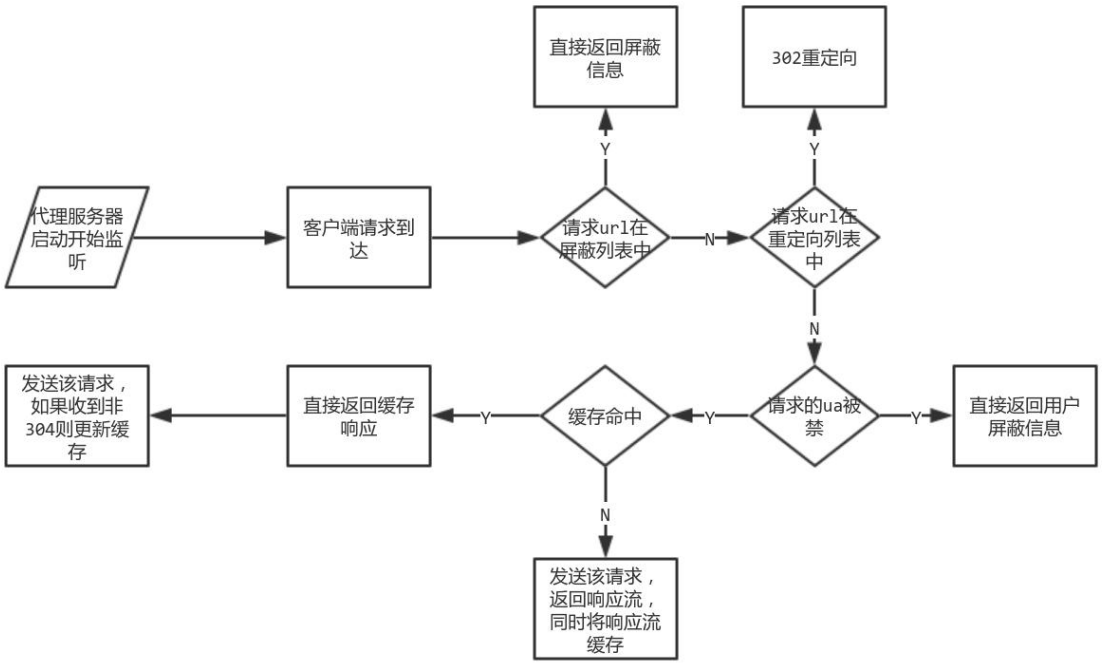
代理服务器，允许一个网络终端（一般为客户端）通过这个服务与另一个网络终端（一般为服务器）进行非直接的连接



首先，代理服务器创建 HTTP 代理服务的 TCP 主套接字，通过该主套接字监听等待客户端的连接请求。当客户端连接之后，读取客户端的 HTTP 请求报文，通过请求行中的 URL，解析客户期望访问的原服务器 IP 地址;创建访问原（目标）服务器的 TCP 套接字，将 HTTP 请求报文转发给目标服务器，接收目标服务器的响应报文，当收到响应报文之后，将响应报文转发给客户端，最后关闭套接字，等待下一次连接。

代理服务器还可以加入缓存机制，当接收到浏览器对远程网站的浏览请求时，代理服务器开始在代理服务器的缓存中检索 URL 对应的对象（网页、图像等对象），找到对象文件后，提取该对象文件的最新被修改时间;代理服务器程序在客户的请求报文首部插入<If-Modified-Since: 对象文件的最新被修改时间>，并向原 Web 服务器转发修改后的请求报文。如果代理服务器没有该对象的缓存，则会直接向原服务器转发请求报文，并将原服务器返回的响应直接转发给客户端，同时将对象缓存到代理服务器中。代理服务器程序会根据缓存的时间、大小和提取记录等对缓存行清理

3.3 HTTP 代理服务器的程序流程图



3.4 HTTP 代理服务器的关键技术及解决方案

HTTP

发送 HTTP 的请求，接受 HTTP 响应需要遵循相应的标准格式

url 屏蔽，钓鱼，用户屏蔽

接受到的请求中包含 url，用户信息（cookie，user-agent 等），通过这些信息可以做过滤处理，钓鱼通过 302 重定向实现

缓存

缓存可以放在硬盘上，也可以放在内存里（如 redis），本程序放在内存中

缓存映射中，请求的 url+method+headers 为键。在 nodejs 的 http.request 中，响应是流的形式，将响应流 pipe 给客户后，该流就结束了，故需将流转成 buffer 保存，缓存命中时再将保存的 buffer 转成流。响应流中只包含响应的 body，故还需保存头部信息。缓存替换机制是最少命中，故缓存映射的值中还需添加命中次数。综上，映射值由响应流的 buffer 形式，响应信息（headers，状态码），命中次数三部分构成。

缓存命中时先返回缓存的响应，再发请求询问是否更新，若状态码不是 304 则更新缓存

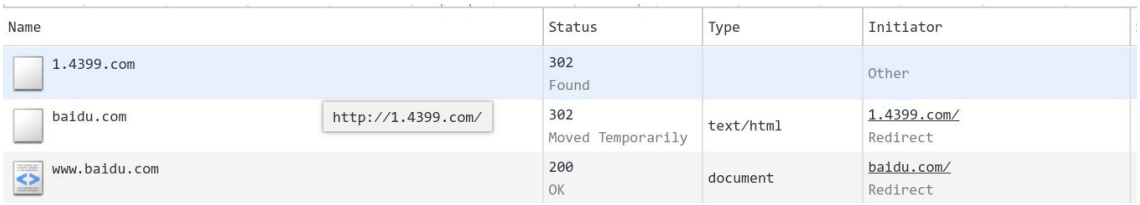
4 实验过程与结果

4.1 屏蔽 url

访问 1.bilabila.cn 返回对应屏蔽信息

4.2 钓鱼

访问 1.4399.com 重定向到 https://baidu.com



4.3 用户屏蔽

user-agent 不包含 chrome，提示 user-agent blocked

4.4 缓存

命中后返回缓存响应流，同时发送头部带 if-modified-since 字段的请求

若响应状态码 304，什么都不做

```
cache hit 2018-04-30T06:50:20.497Z
if-modified-since: Mon, 24 Oct 2011 02:18:30 GMT 304
```

若响应状态码非 304，则更新缓存

```
cache hit 2018-04-30T06:53:26.476Z
if-modified-since: Mon, 30 Apr 2018 06:53:22 GMT 200
cache add:  37 2018-04-30T06:53:26.496Z
cache update 2018-04-30T06:53:26.497Z
```

5 源代码（有详细注释）

```
1. const http = require('http')
2. const url = require('url')
3. const net = require('net')
```

```
const pass = require('stream').PassThrough
const fs = require('fs')
const Duplex = require('stream').Duplex
// no external dependency

const hostname = '127.0.0.1'
const port = '8888'
// block url, block info
const blockList = new Map([
    // ['www.4399.com', 'blocked'],
    ['1.bilabila.cn', 'bilabila blocked']
])
// redirect url
const redirectList = new Map([
    ['1.4399.com', 'https://baidu.com'],
    // ['www.gamersky.com', 'http://bilabila.cn'],
])
// user agent allowed
const uaAllow = ua => ua && ua.toLowerCase().includes('chrome')
// cache response
const cachePool = (size = 20) => {
    const cache = new Map()
    // key=url+method+headers
    const key = req => JSON.stringify({ ...url.parse(req.url),method:req.method,headers:req.headers })
    return {
        set(req, res) {
            // value=[buffer,hit,response]
            const v = [undefined, 1, res]
            const k = key(req)
            cache.set(k, v)
            // stream to buffer
            const data = []
            res.on('data', d => { data.push(d) })
            res.on('end', () => { v[0] = Buffer.concat(data) })
            console.log('cache add: ', cache.size, (new Date).toISOString())
                // remove the least hit
                ; (async () => {
                    if (cache.size > size) {
                        let min = size + 5, index
                        for (let e of cache) {
                            if (e[1][1] < min) {
                                min = e[1][1]
                                index = e[0]
                            }
                        }
                        cache.delete(index)
                    }
                })()
        },
```

```
        get(req) {
            const k = key(req)
            const v = cache.get(k)
            // if buffer defined
            if (v && v[0]) {
                console.log('cache hit', (new Date).toISOString())
                // hit +1
                v[1]++
                // buffer to stream
                const stream = new Duplex()
                stream.push(v[0])
                stream.push(null)
                // stream, cache response
                return [stream, v[2]]
            }
            return []
        }
    }
}
// cache 500 response
const cache = cachePool(500)
//request mode for http
const request = (req, res) => {
    const ua = req.headers['user-agent']
        , u = url.parse(req.url)
        , block = blockList.get(u.hostname)
        , redirect = redirectList.get(u.hostname)
        , allow = uaAllow(ua)
        , [cacheStream, cacheRes] = cache.get(req)
        , opt = {
            hostname: u.hostname,
            port: u.port,
            path: u.path,
            method: req.method,
            headers: req.headers
        }
    if (block) {
        res.end(block)
    } else if (redirect) {
        res.writeHead(302, { 'Location': redirect })
        res.end()
    } else if (!allow) {
        res.end('user-agent blocked')
    } else if (cacheRes) { // cache hit
        // request with if-modified-since
        const lastModified = req.headers['if-modified-since'] || cacheRes.headers['last-modified'] || ''
        opt.headers['if-modified-since'] = lastModified
            http.request(opt, res => {
                console.log('if-modified-since:', lastModified, res.statusCode)
```

```
                if (res.statusCode !== 304) {
                    cache.set(req, res)
                    console.log('cache update', (new Date).toISOString()
)
                }
            }).end()
            if (!cacheRes.headers['last-modified'])
                cacheRes.headers['last-modified'] = (new Date).toUTCStr
ing()
            res.writeHead(cacheRes.statusCode, cacheRes.headers)
            cacheStream.pipe(res, { end: true })
    } else { // cache not hit
            http.request(opt, pRes => {
                if (!pRes.headers['last-modified'])
                    pRes.headers['last-modified'] = (new Date).toUTCStr
ing()
                res.writeHead(pRes.statusCode, pRes.headers);
                cache.set(req, pRes)
                pRes.pipe(res, { end: true })
            }).end()
        }
    }

    http.createServer()
        .on('request', request)
        .listen(port, hostname, () => {
            console.log(`proxy run: ${hostname}:${port}`)
        })
```

6 实验心得

对代理服务器的原理有了更深入的理解

对 nodejs 的 net，http 模块有了实践

更清晰地认识到 http 的安全性问题