

廈門大學



信息学院软件工程系

《计算机网络》实验报告

题 目 实验七 代理服务器软件

班 级 软件工程 2019 级 2 班

姓 名 李世豪

学 号 22920192204229

实验时间 2021 年 5 月 16 日

2021 年 6 月 03 日

填写说明

- 1、本文件为 Word 模板文件，建议使用 Microsoft Word 2019 打开，在可填写的区域中如实填写；
- 2、填表时，勿破坏排版，勿修改字体字号，打印成 PDF 文件提交；
- 3、文件总大小尽量控制在 1MB 以下，勿超过 5MB；
- 4、应将材料清单上传在代码托管平台上；
- 5、在学期最后一节课前按要求打包发送至 cni21@qq.com。

1 实验目的

通过完成实验，掌握基于 RFC 应用层协议规约文档传输的原理，实现符合接口且能和已有知名软件协同运作的软件。

实验任务：

- 1、实现外网代理服务器
- 2、使用 SOCKS4, SOCKS5 协议
- 3、通过 chrome 浏览器和其他浏览器运行代理服务

2 实验环境

操作系统：Mac Big Sur 11.3（类 Unix 系统）

编程语言：C/C++，socket 库

3 实验结果

1. 要实现基于 SOCKS4 和 SOCKS5 的代理服务器，首先必须要对 SOCK4 和 SOCKS5 协议进行详细的了解，连接 SOCK 代理开始时进行的请求响应报文的格式。如下：

SOCKS4 报文请求格式如下（客户端发送请求）：

VN	CD	DSTPORT	DSTIP	USERID	NULL
1	1	2	4	variable	1

- VN是SOCK版本，应该是4；
- CD是SOCK的命令码，1表示CONNECT请求，2表示BIND请求；
- DSTPORT表示目的主机的端口；
- DSTIP指目的主机的IP地址；
- NULL是0；

SOCKS4 服务器响应报文格式如下：

VN	CD	DSTPORT	DSTIP
1	1	2	4

- VN是回应码的版本，应该是0；
- CD是代理服务器答复，有几种可能：
 - 90，请求得到允许；
 - 91，请求被拒绝或失败；
 - 92，由于SOCKS服务器无法连接到客户端的identd（一个验证身份的进程），请求被拒绝；
 - 93，由于客户端程序与identd报告的用户身份不同，连接被拒绝。
- DSTPORT与DSTIP与请求包中的内容相同，但被忽略。

SOCKS5 请求报文和响应报文格式如下：

VER	NMETHODS	METHODS
1	1	1-255

- VER是SOCKS版本，这里应该是0x05；
- NMETHODS是METHODS部分的长度；
- METHODS是客户端支持的认证方式列表，每个方法占1字节。当前的定义是：
 - 0x00 不需要认证
 - 0x01 [GSSAPI](#)
 - 0x02 用户名、密码认证
 - 0x03 – 0x7F由IANA分配（保留）
 - 0x03: 握手挑战认证协议
 - 0x04: 未分派
 - 0x05: 响应挑战认证方法
 - 0x06: [传输层安全](#)
 - 0x07: NDS认证
 - 0x08: 多认证框架
 - 0x09: JSON参数块
 - 0x0A–0x7F: 未分派
 - 0x80 – 0xFE为私人方法保留
 - 0xFF 无可接受的方法

VER	METHOD
1	1

- VER是SOCKS版本，这里应该是0x05；
- METHOD是服务端选中的方法。如果返回0xFF表示没有一个认证方法被选中，客户端需要关闭连接。

之后客户端和服务端根据选定的认证方式执行对应的认证。

认证结束后客户端就可以发送请求信息。如果认证方法有特殊封装要求，请求必须按照方法所定义的方式进行封装。

SOCKS5 通过第一次报文认证后，需要在进行一次请求信息确认，请求报文和响应报文如下（请求有客户端发出，响应由服务器返回）：

SOCKS5请求格式（以字节为单位）：

VER	CMD	RSV	ATYP	DST.ADDR	DST.PORT
1	1	0x00	1	动态	2

- VER是SOCKS版本，这里应该是0x05；
- CMD是SOCK的命令码
 - 0x01表示CONNECT请求
 - 0x02表示BIND请求
 - 0x03表示UDP转发
- RSV 0x00，保留
- ATYP DST.ADDR类型
 - 0x01 IPv4地址，DST.ADDR部分4字节长度
 - 0x03 域名，DST.ADDR部分第一个字节为域名长度，DST.ADDR剩余的内容为域名，没有\0结尾。
 - 0x04 IPv6地址，16个字节长度。
- DST.ADDR 目的地址
- DST.PORT 网络字节序表示的目的端口

VER	REP	RSV	ATYP	BND.ADDR	BND.PORT
1	1	0x00	1	动态	2

- VER是SOCKS版本，这里应该是0x05；
- REP应答字段
 - 0x00表示成功
 - 0x01普通SOCKS服务器连接失败
 - 0x02现有规则不允许连接
 - 0x03网络不可达
 - 0x04主机不可达
 - 0x05连接被拒
 - 0x06 TTL超时
 - 0x07不支持的命令
 - 0x08不支持的地址类型
 - 0x09 – 0xFF未定义
- RSV 0x00，保留
- ATYP BND.ADDR类型
 - 0x01 IPv4地址，DST.ADDR部分4字节长度
 - 0x03域名，DST.ADDR部分第一个字节为域名长度，DST.ADDR剩余的内容为域名，没有\0结尾。
 - 0x04 IPv6地址，16个字节长度。
- BND.ADDR 服务器绑定的地址
- BND.PORT 网络字节序表示的服务器绑定的端口

SOCKS5 支持用户验证，在第一次请求响应后，若服务器返回确认使用用户验证方式进行验证，则先进行以下用户密码认证，格式如下：

SOCKS5 用户名密码认证方式 [\[编辑 \]](#)

在客户端、服务端协商使用用户名密码认证后，客户端发出用户名密码，格式为（以字节为单位）：

鉴定协议版本	用户名长度	用户名	密码长度	密码
1	1	动态	1	动态

鉴定协议版本目前为 0x01。

服务器鉴定后发出如下回应：

鉴定协议版本	鉴定状态
1	1

其中鉴定状态 0x00 表示成功，0x01 表示失败。

以上图片来自维基百科-《SOCKS 协议》

2. 了解 SOCKS 协议的流程之后，根据 SOCKS 运行流程进行编程，如下：

1) Main 函数设计：

main 函数主要包括两个部分，第一部分判断是否置定端口，第二部分是初始化服务器套接字 socket 和死循环运行代理服务。

```
int main(int args, char* argv[]){
    if (args>=1&&argv[1]!=NULL) {
        default_port=atoi(argv[1]);
    }
    else{
        default_port=14301;
    }
    create_TCP_socket();
    printf("完成 Socket 初始化.\n");
    while (true) {
        pthread_connection(NULL);
    }
}
```

2) 主逻辑函数：

每当运行一个进程时就创建另一个进程来处理多个用户连接的情况。每个进程首先接收客户端发送来的连接请求，并建立客户端对于的套接字，每个进程绑定一个客户端。随后进行确认请求报文操作，由函数 Version——confirm 实现，在 version_confirm 函数中同时也进行响应报文执行，执行数据转发程序等一些列主要任务，在完成发送后则返回并关闭当前进程。

```
void* pthread_connection(void*){
    pthread_t pid=pthread_self();
    printf("线程%u 正在运行连接\n",pid);
    struct sockaddr_in clientSockAddr;
    socklen_t len=sizeof(clientSockAddr);
    bzero(&clientSockAddr, len);
}
```

```

        int clientSockFd=accept(proxyServerSock,
        (struct sockaddr*)&clientSockAddr, &len);
        pthread_t thrId;
        pthread_create(&thrId, NULL,
        pthread_connection, NULL);
        char Ip[16]={0};
        int port;
        inet_ntop(AF_INET, &clientSockAddr.sin_addr,
        Ip, len);
        port=ntohs(clientSockAddr.sin_port);
        printf("连接至: %s:%d\n",Ip,port);

        version_confirm(clientSockFd,(struct
        sockaddr*)&clientSockAddr,len);
        printf("确认连接, 线程%u 退出
        \n",pthread_self());
        pthread_kill(pid, 0);
        return NULL;
    }

```

3) Version_confirm 函数:

首先确认 SOCKS 请求版本, 不同版本对应不同处理分支, SOCK4 版本不支持用户验证功能, 因此在第一次确认请求后即可进行连接返回信息, 并开始转发数据; SOCK5 支持用户验证, 因此在第一次确认认证方法后, 需要先进行验证, 再由客户端发送目标信息的请求报文, 由代理服务器进行数据转发。

```

void version_confirm(int fd,struct sockaddr*
clientSockAddr,socklen_t len){
    char comfirmInfo[16]={0};
    bzero(&comfirmInfo, sizeof(comfirmInfo));
    read(fd, comfirmInfo, sizeof(comfirmInfo));
    Ver=comfirmInfo[0];

    if (Ver==Ver4) {
        if (Ver4_Socks(comfirmInfo,fd)) {
            response_sock4 responseInfo;
            bzero(&responseInfo,
            sizeof(responseInfo));
            memcpy(&responseInfo, comfirmInfo,
            sizeof(response_sock4));

            //get server address.
            struct sockaddr_in original_server_addr;
            bzero(&original_server_addr,
            sizeof(sockaddr_in));
            original_server_addr.sin_family=AF_INET;
            memcpy(&original_server_addr.sin_addr,
            &responseInfo.ip, sizeof(responseInfo.ip));

```

```

        memcpy(&original_server_addr.sin_port,
&responseInfo.port, sizeof(responseInfo.port));

        //complete the parameters concrete.
start to connect the original server.
        int original_server_fd=socket(AF_INET,
SOCK_STREAM, 0);
        if (original_server_fd==-1) {
            printf("create socket failed.\n");
            return;
        }

        int ret=connect(original_server_fd,
(struct sockaddr*)&original_server_addr,
sizeof(original_server_addr));

        if (ret==-1) {
            printf("connection to original server
fail.\n");
            return;
        }

        printf("connect to original server
success.\n");

        //sock4 start transfer data once it pass
the Authorize, without sending back response that
successfully connect to the original server.
        //start get info and transfer info to
client.

        sock_Transfer(fd,original_server_fd);
    }

}
else if(Ver==Ver5){
    if (Ver5_Socks(confirmInfo,fd)) {
        char connection_request[100]={0};
        bzero(connection_request,
sizeof(connection_request));
        read(fd, connection_request, 100);

        connect_request request_1;
        bzero(&request_1, sizeof(request_1));
    }
}

```



```

        memcpy(&request_1, &connection_request,
sizeof(connect_request));

        if (request_1.cmd!=Connect) {
            printf("only support connect
cmd.\n");
            return;
        }

        if
(request_1.type!=IPv4&&request_1.type!=Domain) {
            printf("only support IPv4 or Domain
protocol.\n");
            return;
        }

        //get server address.
        struct sockaddr_in original_server_addr;
        bzero(&original_server_addr,
sizeof(sockaddr_in));
        original_server_addr.sin_family=AF_INET;
        if (request_1.type==IPv4) {
            memcpy(&original_server_addr.sin_addr
.s_addr, &request_1.address, 4);
            memcpy(&original_server_addr.sin_port
, &request_1.port, 2);
        }
        else{
            int
dormainLen=request_1.address[0]&0xff;
            char hostname[dormainLen];
            for (int i=0; i<dormainLen; i++) {
                hostname[i]=connection_request[i+
5];
            }
            char port[2]={0};
            port[0]=connection_request[dormainLen
+5];
            port[1]=connection_request[dormainLen
+6];

            hostent*
host=gethostbyname(hostname);
            if (host==NULL) {
                printf("Get host address
fail.\n");
                return;
            }

```

```

        }
        memcpy(&original_server_addr.sin_addr
.s_addr, host->h_addr, host->h_length);
        memcpy(&original_server_addr.sin_port
, port, 2);
    }

    //complete the parameters concrete.
    start to connect the original server.
    int original_server_fd=socket(AF_INET,
SOCK_STREAM, 0);
    if (original_server_fd==-1) {
        printf("create socket faild.\n");
        return;
    }

    int ret=connect(original_server_fd,
(struct sockaddr*)&original_server_addr,
sizeof(original_server_addr));

    if (ret==-1) {
        printf("connection to original server
fail.\n");
        return;
    }

    printf("connect to original server
success.\n");

    //now connection is done. proxyServer
    response the client that it connect to the original
    server successfully.
    char connect_res[10]={0};
    bzero(connect_res, sizeof(connect_res));

    connect_response connect_resp;
    bzero(&connect_resp,
sizeof(connect_resp));
    connect_resp.ver=0x05;
    connect_resp.rep=0x00;
    connect_resp.rsv=0x00;
    connect_resp.type=0x01;
    memcpy(&original_server_addr.sin_addr.s_a
ddr, &request_1.address, 4);
    memcpy(&original_server_addr.sin_port,
&request_1.port, 2);

```

```

        memcpy(&connect_resp.address,&request_1.a
ddress , 4);
        memcpy(&connect_resp.port,
&request_1.port, 2);

        memcpy(connect_res, &connect_resp,
sizeof(connect_resp));
        ret=write(fd, &connect_res, 10);
        if (ret==-1) {
            printf("fail to send response.\n");
        }
        printf("complete to send response.\n");

        //start get info and transfer info to
client.

        sock_Transfer(fd,original_server_fd);
    }
}
else{
    printf("version not permit or request info
wrong or response send fails.\n");
    return;
}

}

```

4. 转发数据功能实现:

在完成认证后, SOCKS 服务器就开始进行代理转发数据, SOCKS 不会对请求目标报文进行确认, 而直接转发到目标服务器, 再将从目标服务器接收到的目标数据转发到客户端。

同时这里不使用 accept 的阻塞通信模式, 而是选择使用 linux 上的 Select 非阻塞通信模式, 使用方法主要使用 Select 函数对 SOCKET 文件描述符集合进行活动判断, 即当目标服务器或客户端准备就绪时(有写入和读出 I/O 需求时, 通过 SOCKET_FD 表现), Select 函数返回值 > 1 的正数, 此时再由 FD_ISSET 宏进行活动判断, 从而对请求就绪的 SOCKET 进行通信和传输信息。

```

void sock_Transfer(int client_fd,int original_server_fd){
    printf("线程%u 正在转发数据\n",pthread_self());
    char recv_buff[1024*512]={0};
    fd_set allsocket;
    struct timeval time_out;//最大时延设置为 15 秒
    time_out.tv_sec=15;
    time_out.tv_usec=0;

```

```

while (1) {
    FD_ZERO(&allsocket);
    FD_SET(client_fd,&allsocket); //加入客户端套接字
    FD_SET(original_server_fd,&allsocket); //加入初始服务器
套接字

    //进行套接字选择, 对有响应的套接字进行对应处理, 客户端接收请
    求报文, 初始服务器端请求对应目标数据, 并进行转发到初始套接字
    int
    ret=select(client_fd>original_server_fd?client_fd+1:origina
    l_server_fd+1, &allsocket, NULL, NULL, NULL);
    //select 返回 0 代表无响应, select 返回-1 代表套接字出错, 一
    般为中断连接; 返回>0 代表有响应, 并把有响应的套接字加入到 fd_read 数组
    中

    if (ret==0) {
        continue;
    }
    else if (ret==-1){
        break;
    }
    //处理有响应的 socket
    //处理客户端请求数据报文
    if (FD_ISSET(client_fd,&allsocket)) {
        memset(recv_buff, 0, sizeof(recv_buff));
        //接收请求报文
        ret=recv(client_fd, recv_buff,
sizeof(recv_buff), 0);
        if (ret<=0) {
            break;
        }
        else{
            //转发请求信息
            ret=write(original_server_fd, recv_buff,
ret);

            if (ret==-1) {
                break;
            }
        }
    }
    //对服务器发来的信息进行转发
    else if(FD_ISSET(original_server_fd,&allsocket)){
        memset(recv_buff, 0, sizeof(recv_buff));
        //接受目标信息

```

```

        ret=recv(original_server_fd, recv_buff,
sizeof(recv_buff), 0);
        if (ret<=0) {
            break;
        }
        else{
            //转发目标信息
            ret=write(client_fd, recv_buff, ret);
            if (ret==-1) {
                break;
            }
        }
    }
}
return;
}
}

```

5. 运行效果验证:

运行方式如下:

第一参数为 6000, 即设置代理窗口为 6000

若未输入参数, 则默认使用端口 14301

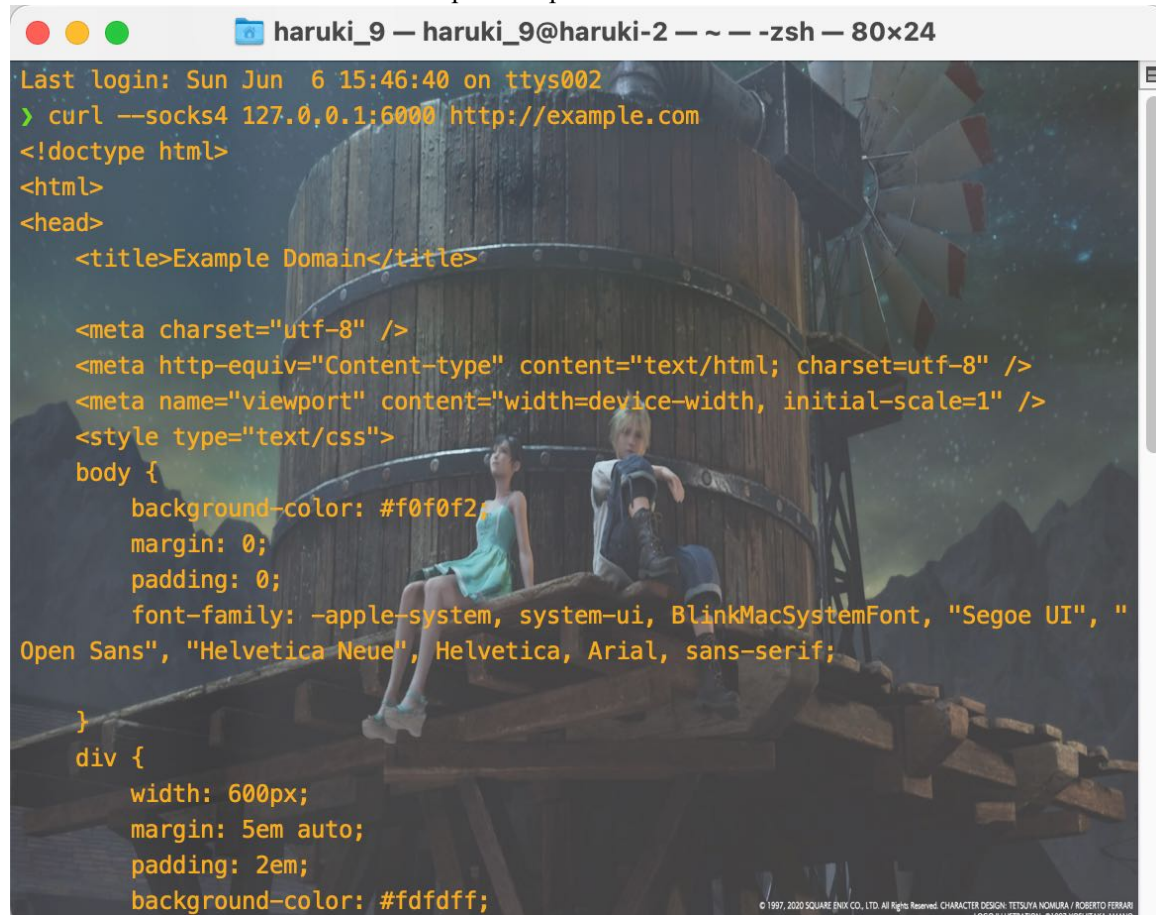


这里因为某些原因 (本人使用的是 MAC 系统, 同时本人并没有购买适配 ARM 64M1 的 Mac 端上的虚拟机运行软件), 因此这里不使用虚拟机进行模拟测试, 而使用本机进行测试, 同时由于时间的不足, 这里虽然实现了 SOCKS5 用户名密码认证方式, 但是并没有进行对应测试, 因此不知道是否会运行出错。以下为对 SOCKS4 和 SOCKS5 代理的测试, 测试网址包括 <http://example.com> 和 www.baidu.com, 如下:

1) SOCK4 代理

首先是在本机系统（Mac）上使用 curl 指令进行 sock4 代理验证：

指令：`curl --socks4 127.0.0.1:6000 http://example.com`



```
haruki_9 — haruki_9@haruki-2 — ~ — -zsh — 80x24
Last login: Sun Jun  6 15:46:40 on ttys002
> curl --socks4 127.0.0.1:6000 http://example.com
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <style type="text/css">
    body {
      background-color: #f0f0f2;
      margin: 0;
      padding: 0;
      font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "
Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
    }
    div {
      width: 600px;
      margin: 5em auto;
      padding: 2em;
      background-color: #fdfdff;
```



```
Debug — sudo ./Proxy 6000 — ./Proxy — Proxy < sudo — 80x24
[Password:
完成Socket初始化.
线程52346176正在运行连接
连接至: 127.0.0.1:53112
线程1829433344正在运行连接
Cmd-type: Connect.
connect to original server success.
线程52346176正在转发数据
确认连接, 线程52346176退出
线程52346176正在运行连接
连接至: 127.0.0.1:53114
线程1830006784正在运行连接
Cmd-type: Connect.
connect to original server success.
线程1829433344正在转发数据
确认连接, 线程1829433344退出
连接至: 127.0.0.1:53118
线程1830580224正在运行连接
Cmd-type: Connect.
connect to original server success.
线程52346176正在转发数据
确认连接, 线程52346176退出
线程52346176正在运行连接
```

在 chrome 浏览器中代理 SOCKS4，使用 SwitchyProxy 发送 SOCKS4 请求进行确认。

未开启代理服务器时：

ACTIONS

⌚ 应用选项

⌚ 撤销更改

 情景模式: proxy

⌚ 导出PAC

📄 更改名称

🗑 删除

代理服务器

网址协议	代理协议	代理服务器	代理端口	
(默认)	SOCKS4 ▾	127.0.0.1	6000	🔒

▾ 显示高级设置

不代理的地址列表

不经过代理连接的主机列表: (每行一个主机)

(可使用通配符等匹配规则...)



未连接到互联网

代理服务器出现问题，或者地址有误。

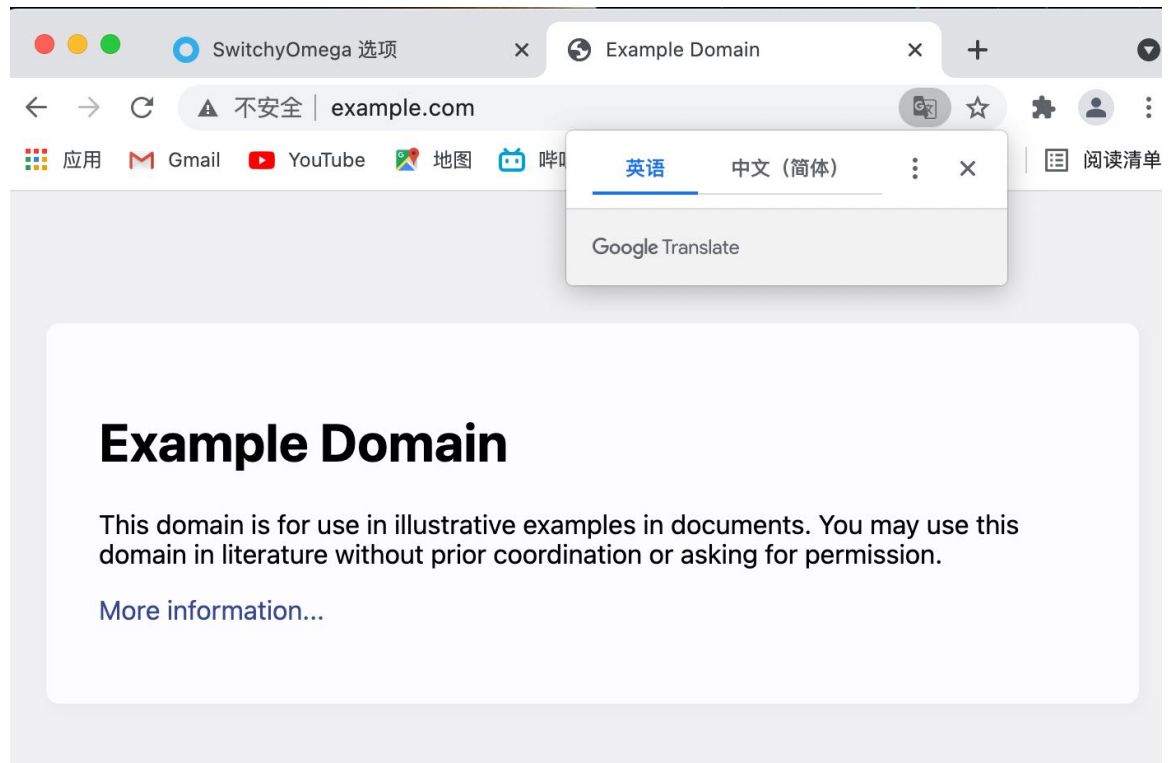
请试试以下办法：

- 联系系统管理员
- [检查代理服务器地址](#)

ERR_PROXY_CONNECTION_FAILED

详细信息

开启代理服务器后：



下面测试 www.baidu.com

```
curl --socks4 127.0.0.1:6000 http://www.baidu.com
```

```
haruki_9 — haruki_9@haruki-2 — ~ — -zsh — 80x24
Last login: Sun Jun  6 15:53:31 on ttys001
> curl --socks4 127.0.0.1:6000 http://www.baidu.com
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html;cha
rset=utf-8><meta http-equiv=X-UA-Compatible content=IE=Edge><meta content=always
name=referrer><link rel=stylesheet type=text/css href=http://s1.bdstatic.com/r/
www/cache/bdorz/baidu.min.css><title>百度一下，你就知道</title></head> <body lin
k=#0000cc> <div id=wrapper> <div id=head> <div class=head_wrapper> <div class=s_
form> <div class=s_form_wrapper> <div id=lg> <img hidefocus=true src=//www.baidu
.com/img/bd_logo1.png width=270 height=129> </div> <form id=form name=f action=/
/www.baidu.com/s class=fm> <input type=hidden name=bdorz_come value=1> <input ty
pe=hidden name=ie value=utf-8> <input type=hidden name=f value=8> <input type=hi
dden name=rsv_bp value=1> <input type=hidden name=rsv_idx value=1> <input type=h
idden name=tn value=baidu><span class="bg s_ipt_wr"><input id=kw name=wd class=s
_ipt value maxlength=255 autocomplete=off autofocus></span><span class="bg s_btn
_wr"><input type=submit id=su value=百度一下 class="bg s_btn"></span> </form> </
div> </div> <div id=u1> <a href=http://news.baidu.com name=tj_trnews class=mnav>
新闻</a> <a href=http://www.hao123.com name=tj_trhao123 class=mnav>hao123</a> <a
href=http://map.baidu.com name=tj_trmap class=mnav>地图</a> <a href=http://v.ba
idu.com name=tj_trvideo class=mnav>视频</a> <a href=http://tieba.baidu.com name=
tj_trtieba class=mnav>贴吧</a> <noscript> <a href=http://www.baidu.com/bdorz/log
in.gif?login&tpl=mn&u=http%3A%2F%2Fwww.baidu.com%2f%3fbdorz_come%3d1 nam
e=tj_login class=lb>登录</a> </noscript> <script>document.write('<a href="http:/
```



2) 下面进行 SOCKS5 协议代理测试:

命令: `curl --socks5 127.0.0.1:6000 http://example.com`

```
haruki_9 — haruki_9@haruki-2 — ~ — -zsh — 80x24

> curl --socks5 127.0.0.1:6000 http://example.com
<!doctype: html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <style type="text/css">
    body {
      background-color: #f0f0f2;
      margin: 0;
      padding: 0;
      font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "
Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
    }
    div {
      width: 600px;
      margin: 5em auto;
      padding: 2em;
      background-color: #fdfdff;
      border-radius: 0.5em;
    }
  </style>
</head>
<body>
  <div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use
    </p>
  </div>
</body>
</html>
```

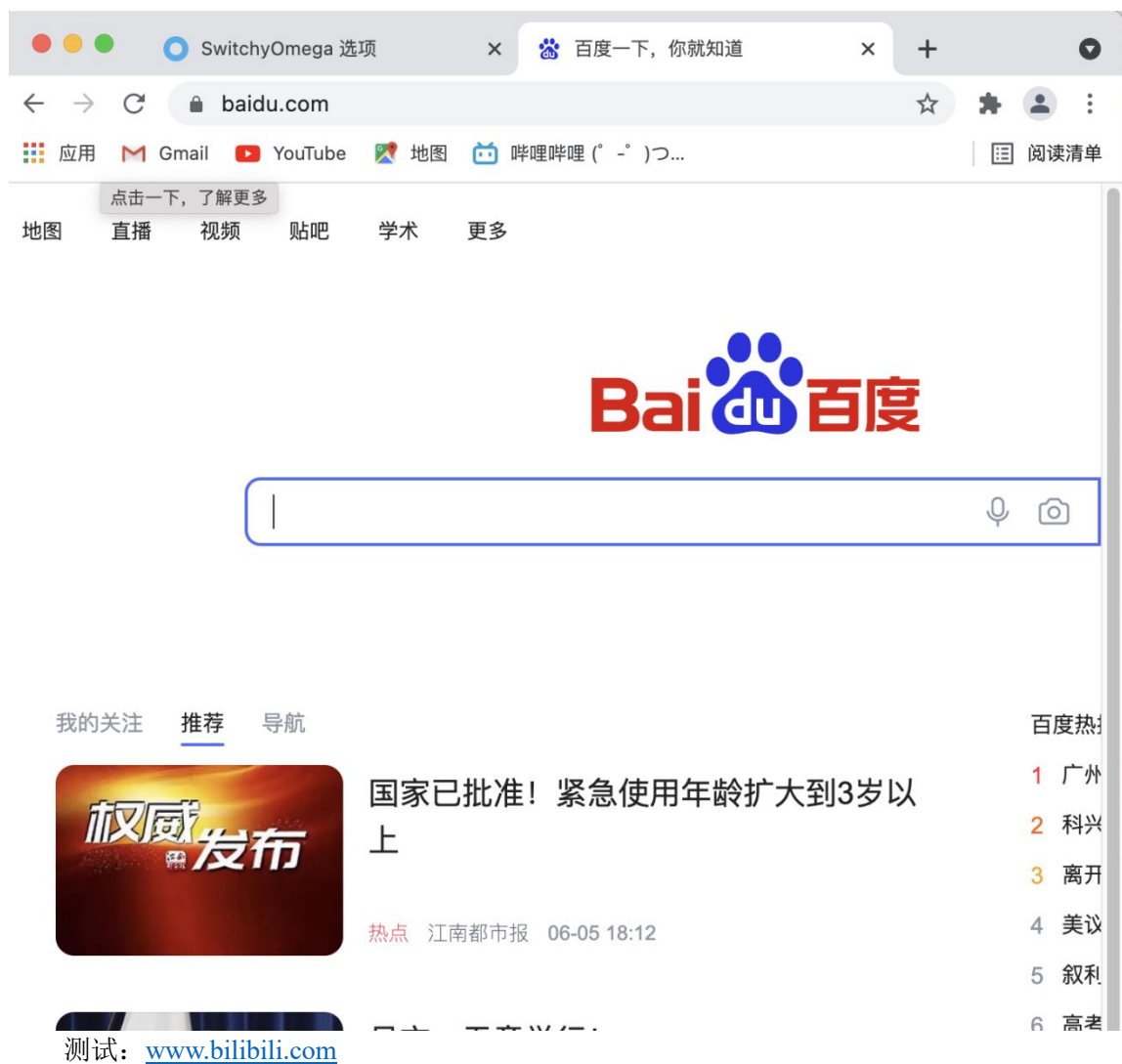
ACTIONS

☒ 应用选项☐ 撤销更改**情景模式：** proxy

代理服务器

网址协议	代理协议	代理服务器	代理端口	
(默认)	SOCKS5	127.0.0.1	6000	
显示高级设置				

不代理的地址列表





4 实验代码

本次实验的代码已上传于以下代码仓库：代码放置于 Github，地址如下：

Github: https://github.com/Haruki9/Computer-Network_Labs/tree/main

Gitee: https://gitee.com/haruki9/computer-network_-labs

5 实验总结

本次实验让我对代理服务器的运行流程有了一个深切的理解与体会,对 SOCKS 协议有了全面了了解,同时也加深了我对 protocol 一词的理解,对应用层网络协议在实践中得到学习。