Socket应用编程实验报告

李昊宸

2017K8009929044

（一）基于Socket API的HTTP服务器、客户端实验

一、实验内容

1. 使用C语言分别实现最简单的HTTP服务器和HTTP客户端

服务器监听80端口，收到HTTP请求，解析请求内容，回复HTTP应答

对于本地存在的文件，返回HTTP 200 OK和相应文件

对于本地不存在的文件，返回HTTP 404 File Not Found

2. 服务器、客户端只需要支持HTTP Get方法，不需要支持Post等方法

3．服务器使用多线程支持多路并发

二、实验流程

1. 搭建实验环境

实验用到的脚本的两个版本分别放在两个文件夹内：只支持单连接的版本non-multipid和支持多连接的版本multipid。每个版本内都有服务器端echo-server、客户端echo-client、Makefile和待发送的测试文件1.txt，接受的文件命名为1\_receive.txt。具体代码见后续实验分析。（这里关于服务器和客户端的取名在写代码时忘记修改了，问题不大）

2.启动脚本

1)服务器、客户端功能测试

cd non-multipid

sudo python topo.py

mininet> xterm h1 h2

h1# ./echo\_server

h2# ./echo\_client

h2# enter message: 1 //表示向http://10.0.0.1 获取文件1.txt，可重复多次测试客户端的多次获取文件性能

h2# enter message: 2 //表示向http://10.0.0.1 获取文件2.txt，可重复多次测试客户端的多次获取文件性能

因为10.0.0.1只有文件1.txt，所以当获取1.txt时会向客户端发送HTTP/1.0 200 OK

报文，获取2.txt时会向客户端发送HTTP/1.0 404 FILE NOT FOUND报文

2)服务器、wget功能测试

cd non-multipid

sudo python topo.py

mininet> xterm h1 h2

h1# ./echo\_server

h2# wget http://10.0.0.1/1.txt

h2# wget http://10.0.0.1/2.txt

因为10.0.0.1只有文件1.txt，所以当获取1.txt时会向wget端发送HTTP/1.0 200 OK

报文，获取2.txt时会向wget端发送HTTP/1.0 404 FILE NOT FOUND报文

3)python HTTP、客户端功能测试

cd non-multipid

sudo python topo.py

mininet> xterm h1 h2

h1# python -m SimpleHTTPServer 80

h2# ./echo-client

h2# enter message: 1 //表示向http://10.0.0.1 获取文件1.txt

h2# enter message: 2 //表示向http://10.0.0.1 获取文件2.txt

因为10.0.0.1只有文件1.txt，所以当获取1.txt时会向客户端发送HTTP/1.0 200 OK

报文，获取2.txt时会向客户端发送HTTP/1.0 404 FILE NOT FOUND报文

4)服务器、客户端并发功能测试

cd multipid

sudo python topo.py

mininet> xterm h1 h2 h3

h1# ./echo-server

h2# ./echo-client

h3# ./echo-client

h2# enter message:1

h3# enter message:2

h2# enter message:2

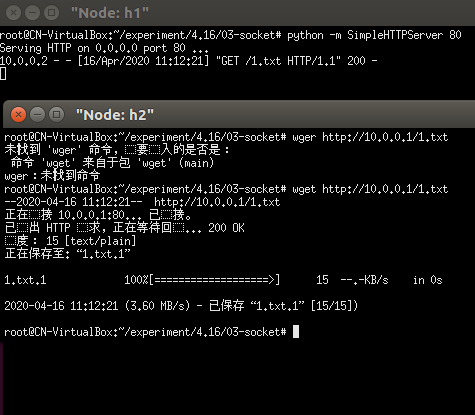
h3# enter message:1

这里交叉验证的目的是，如果h2和h3可以同时与h1保持连接，并且进行数据传送，

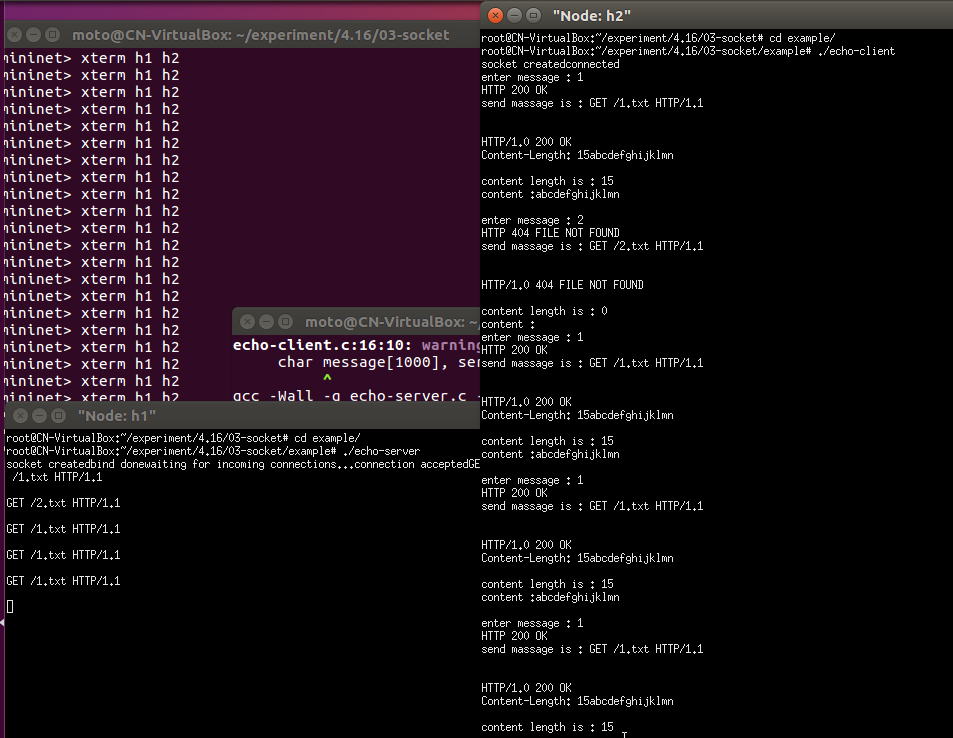
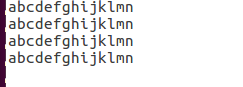
那么服务器就具备多线程/多进程支持并发访问的功能。

三、实验结果及分析

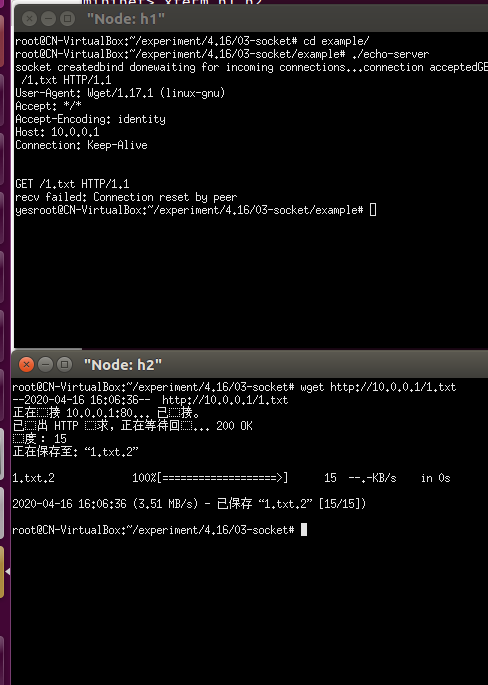
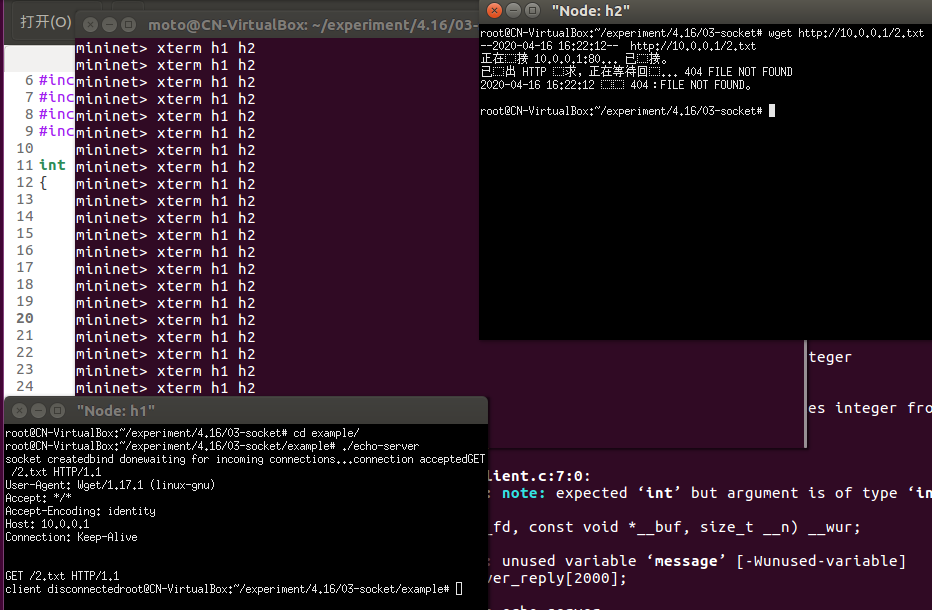
1. 实验结果

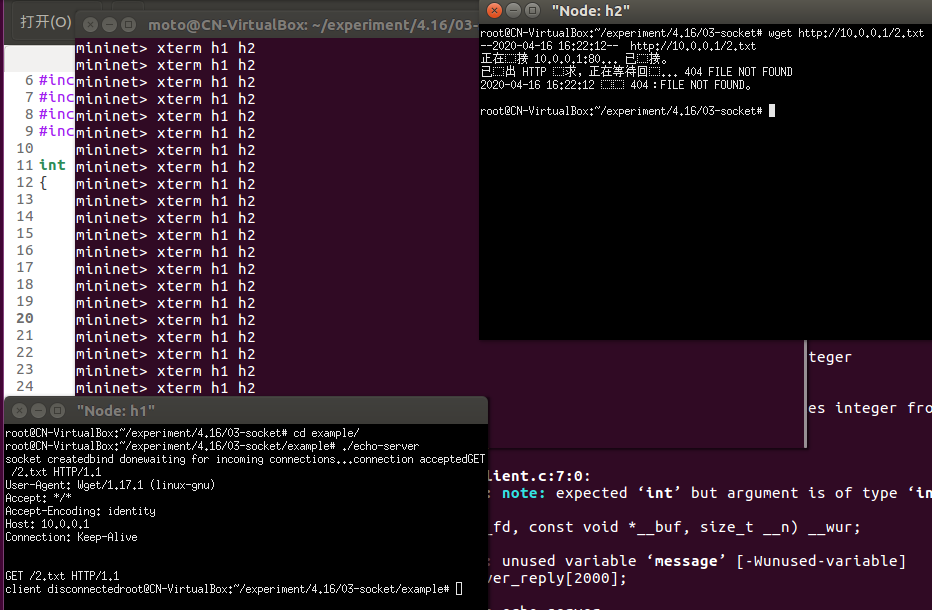


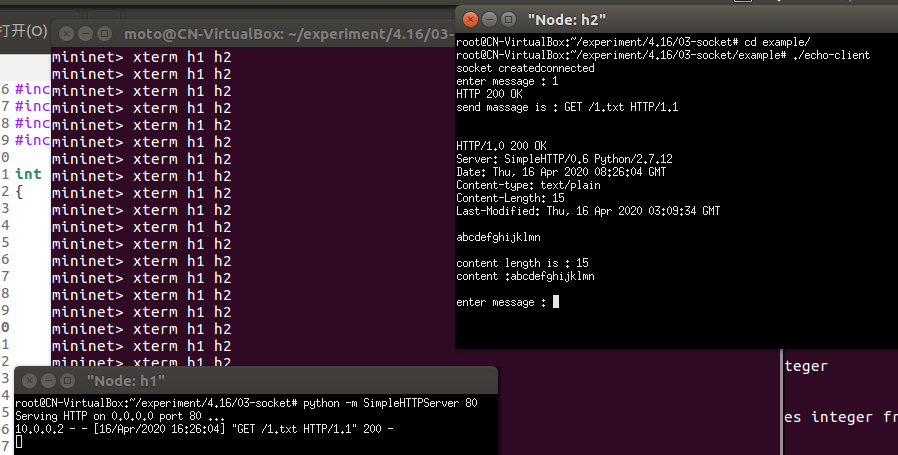
图一 使用python HTTPServer和wget进行数据传输示例

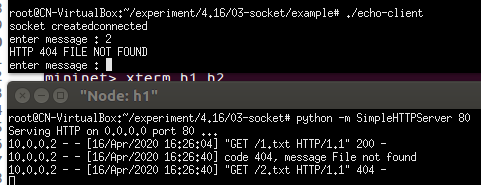


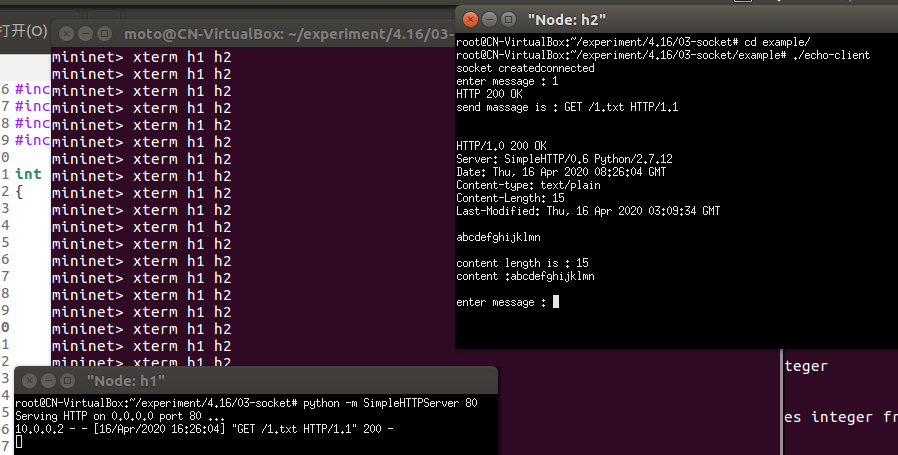
图二 服务器、客户端之间进行连续传输测试 及 客户端连续接收的数据（为方便测试就保存至同一文件）

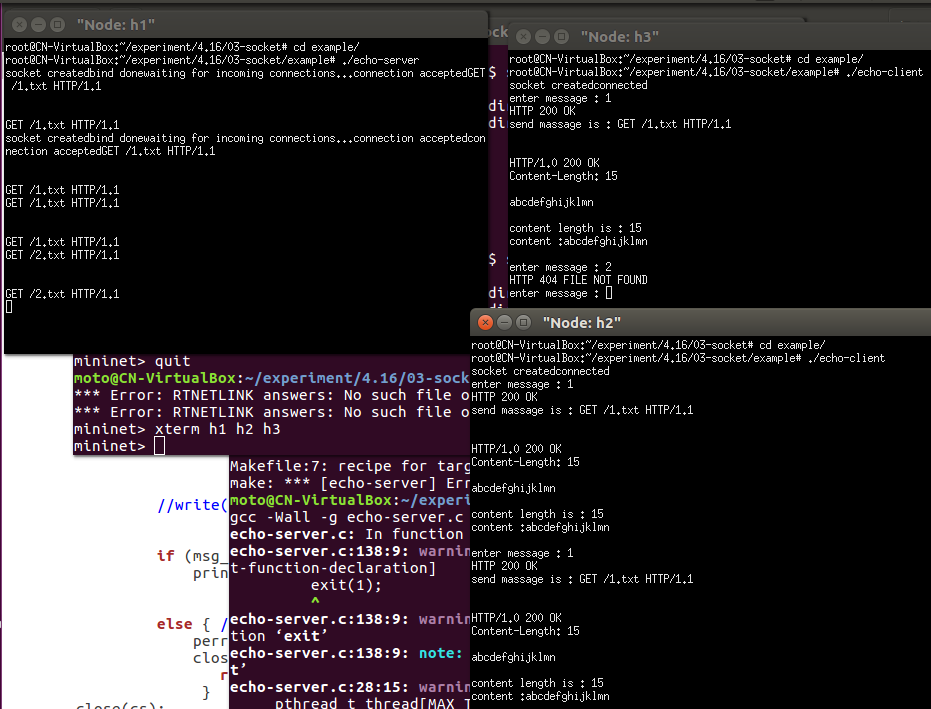
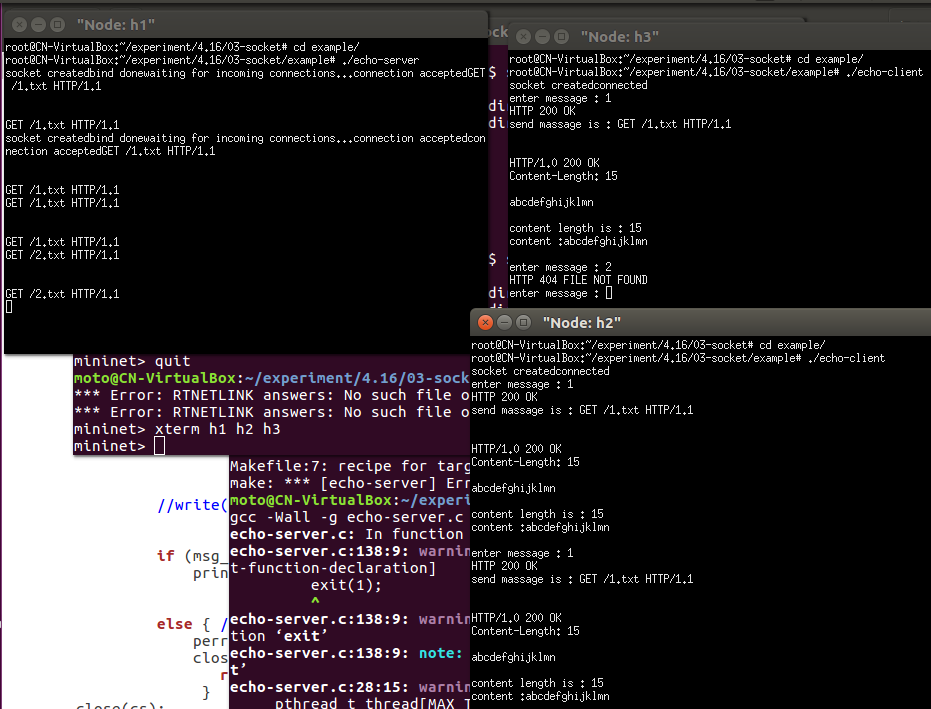


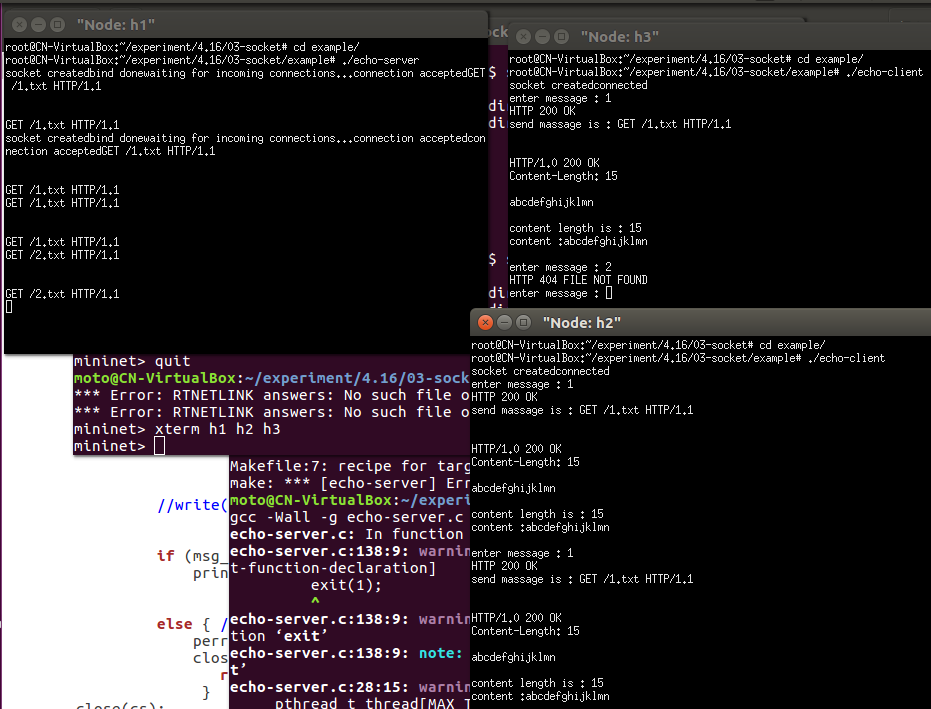


图三 使用wget测试服务器性能 左图请求有效文件 右图请求无效文件





图四 使用python HTTPServer测试客户端性能 左图请求有效文件 右图请求无效文件



图五 测试服务器的并发访问（h1为服务器，h2、h3为并发的客户端）

2. 实验分析

HTTP报文：用于 HTTP 协议交互的信息被称为 HTTP 报文。请求端（客户端）的 HTTP 报文叫做请求报文，响应端（服务器端）的叫做响应报文。

1）请求报文

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 请求方法 | 空格 | URL | 空格 | 协议版本 | \r | \n | 请求行 |
| 头字段名 | ： | 空格 | value | \r | \n |  | 请求头 |
| …… | | | | | |  |
| 头字段名 | ： | 空格 | value | \r | \n |  |
| \r | \n |  |  |  |  |  |  |
| data | | | | | |  | 请求体 |

请求方法：

OPTIONS 使服务器回传URL指定的资源所支持的HTTP请求方法

GET 向服务器请求获取URL指定的资源。GET报文没有请求体

HEAD 与GET类似，但是该方法只获取某个资源的元数据，响应报文中不包含资源本身

POST 向URL指定的资源提交数据，数据就是请求体中的数据

PUT 向URL指定的位置上传资源

DELETE 请求服务器删除URL指定的资源

TRACE 有点像echo，原来的请求报文会包含在响应报文的响应体中

URL：目标资源的位置。只包含一个完整的文件路径和查询字符串，而不包含协议、主机名和端口号。

协议版本：HTTP/1.0 HTTP/1.1

HTTP1.0对于每个连接都只能传送一个请求和响应，请求就会关闭，HTTP1.0没有Host字段;而HTTP1.1在同一个连接中可以传送多个请求和响应，多个请求可以重叠和同时进行，HTTP1.1必须有Host字段。

头字段名：

Accept 客户端可接收资源的类型

Accept-Charset 客户端可接收的字符集

Accept-Encoding 客户端可接收的数据压缩类型

Accept-Languague 客户端可接受的语言

Authorization 授权信息

Cache-Control 本次请求和响应的缓存机制

Connection 处理完本次请求后是继续保持连接（这里的值是Keep-Alive或协

议是HTTP/1.1）还是断开连接

Cookie 发送给服务器的cookie

Content-Type 请求体数据的类型

Content-Length 请求体数据的大小，单位为字节

Host 请求的网站域名

Range 请求资源的某一部分，例：Range: bytes=328-421

Referer 来源网页的地址

User-Agent 版本信息

请求体：

如果方法字段是GET，就为空

如果方法字段是POST，就在此存放要提交的数据

2）响应报文

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 协议版本 | 空格 | 状态码 | 空格 | 描述 | \r | \n | 响应行 |
| 头字段名 | ： | 空格 | value | \r | \n |  | 响应头 |
| …… | | | | | |  |
| 头字段名 | ： | 空格 | value | \r | \n |  |
| \r | \n |  |  |  |  |  |  |
| data | | | | | |  | 响应体 |

协议版本：HTTP/1.0 HTTP/1.1

状态码： 描述：

100-199 成功接受请求，要求客户端继续提交请求以完

成整个处理过程

200 OK 请求成功，所请求的内容与响应报文一起返回

301 Moved Permanently 请求的资源已被永久移动到新的位置

302 Found 请求的资源现在临时移到新的位置

304 Not Modified 所请求的资源和上次请求没有变化，要求客户

端使用之前的缓存

400 Bad Request 请求参数或语义有错，服务器无法理解

401 Unauthorize 请求需要验证用户

403 Forbidden 服务器拒绝访问

404 Not Found 服务器上没有所请求的资源

408 Request Timeout 请求超时

500 Internal Server Error 未知错误

503 Service Unavailable 服务器暂时不可用

头字段名：

Allow 对资源请求所允许的方法

Cache-Control 告知客户端是否可以缓存该资源

Content-Encoding 响应体数据压缩方式

Content-Length 响应体数据大小，单位为字节

Content-Location 所请求资源的候选地址

Content-Type 响应体数据的类型

Date 当地的GMT时间

Last-Modified 所请求资源最后被修改的时间

Location 将请求重定向的地址，配合302使用

Server 服务器的信息

响应体：

所请求资源的数据

3)GET方法和POST方法的对比

提交数据的形式：

GET请求的数据会附在URL之后(就是把数据放置在HTTP协议头中)，会直接展现在地址栏中，以?分割URL和传输数据，参数之间以&相连，如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5 %A5%BD。

如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，

得出如：%E4 %BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII。

而POST方法则会把数据放到请求数据字段中以&分隔各个字段，请求行不包含数据参数，地址栏也不会额外附带参数

提交数据的大小：

get方法提交数据的大小直接影响到了URL的长度，但HTTP协议规范中其实是没有对URL限制长度的，限制URL长度的是客户端或服务器的支持的不同所影响：比如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

post方式HTTP协议规范中也没有限定，起限制作用的是服务器的处理程序的处理能力。

所以大小的限制还是得受各个web服务器配置的不同而影响。

提交数据的安全：

POST比GET方式的安全性要高

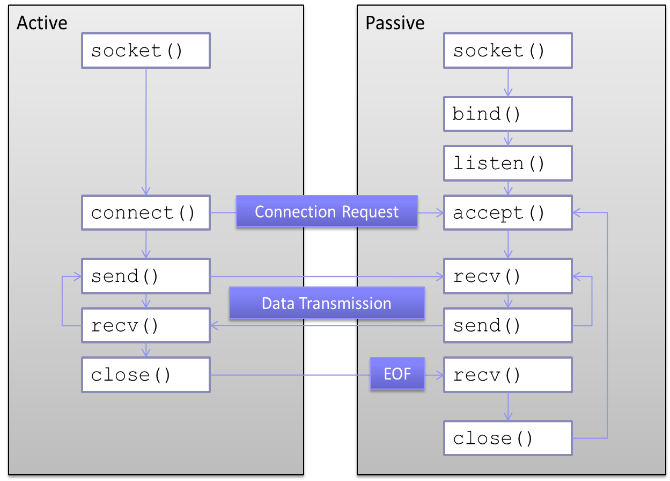
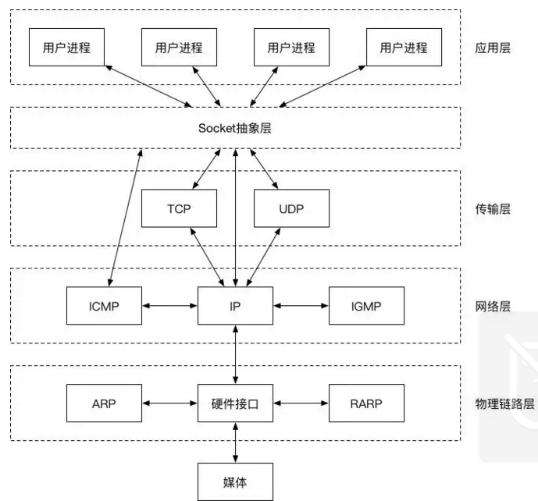
通过GET提交数据，用户名和密码将明文出现在URL上，因为一下几个原因get方式安全性会比post弱：

(1)登录页面有可能被浏览器缓存

(2)其他人查看浏览器的历史纪录，那么别人就可 以拿到你的账号和密码了

(3)当遇上跨站的攻击时，安全性的表现更差了

Socket API：Socket，套接字，是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口，供应用层调用实现进程在网络中的通信。



图六 socket所在位置（左图） socket通信结构（右图）

Socket调用流程：

1）建立socket文件描述符

数据收发两端都需要建立socket文件描述符

int socket(int domain, int type, int protocol);

Domain: AF\_INET

Type：

SOCK\_STREAM TCP

SOCK\_DGRAM UDP

Protocol: 0

return: socket文件描述符

例：int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

2)将socket文件描述符与监听地址绑定

只需要被动建立连接一方进行绑定（bind）

int bind(int sockfd, const struct sockaddr \*addr,

socklen\_t addrlen);

sockfd: Socket文件描述符

addr: 需要绑定的地址和端口

addrlen: 地址和端口数据结构的长度

return： bind

例：struct sockaddr\_in server;

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = INADDR\_ANY; //INADDR\_ANY表示监听所有地址

server.sin\_port = htons(80);

bind(sockfd, (struct sockaddr \*)&server, sizeof(server));

网络字节序与本地字节序：网络字节序使用Big Endian（低地址存放数值的高位，大端序），Intel x86平台使用Little Endian（低地址存放数值的地位，小端序），故在发送整数数据时需要转换字节序（short，int，long，long long等），而收发字符串/字节流时不需要关心。

网络->本地主机：ntohs()，network to host short，16位为单位

ntohl()，network to host long，32位为单位

ntohll(),network to host long long，64位为单位

本地主机->网络：htons()，host to network short，16位为单位

htonl()，host to network long，32位为单位

htonll(),host to network long long，64位为单位

3)进行监听

只在被动建立连接一方进行监听，等待新的连接请求

int listen(int sockfd, int backlog)

sockfd: 之前建立的socket文件描述符

backlog: 可以理解为待处理的最大连接数目

例：listen(sockfd，128)；

4）接受连接请求

被动建立连接一方需要显式的接受连接请求

int accept(int sockfd, struct sockaddr \*addr, socklen\_t

\*addrlen);

sockfd: 之前建立的socket文件描述符

addr: 用于存储对端网络地址的数据结构

addrlen: 指定addr大小

return：该连接对应的文件描述符，以后收发数据都使用该文件描述符

例：int csock = accept(sockfd, (struct sockaddr\_in \*)&caddr, &clen);

5）申请建立连接

连接的另一方需要主动建立连接

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

sockfd: 之前建立的socket文件描述符

addr: 对端的网络地址（包括IP地址和端口号）

addrlen: addr的大小（不同协议的addr大小不同）

return：指示连接是否建立成功

例：struct sockaddr\_in dst;

dst.sin\_family = AF\_INET;

dst.sin\_addr.s\_addr = inet\_addr("10.0.0.1");

dst.sin\_port = htons(12345);

connect(sock, (struct sockaddr \*)&dst, sizeof(dst));

6）数据传输

数据发送方

int send(int sockfd, const void \*buf, size\_t len, int flags);

数据接收方

int recv(int sockfd, void \*buf, size\_t len, int flags);

连接任意一端都可以发送或者接收数据

对于一个阻塞式(blocking) socket：

send直到所有数据被拷贝到协议栈缓存中才返回，或产生错误返回

recv直到接收到数据(不一定等于len)，或对端关闭连接（返回0），或产生错

误才返回

另外，对于recv的一方，也可以通过write方法直接操作socket描述符达到send的效果。

具体操作过程见实验代码详解。

（二）实验代码详解

一、服务器端

1）读取要传输的文件

char \*ok = "HTTP/1.0 200 OK\r\nContent-Length: 15\r\n\r\n";

char OK[2000] = {0};

char buf[1000] = {0};

//read file

FILE \*fd;

if(!(fd = fopen("1.txt","r")))

{

printf("open 1.txt failed!\n");

return 1;

}

int m = 0;

while((x = fgetc(fd))!= EOF)

{

buf[m] = x;

m++;

}

strcat(OK,ok);

strcat(OK,buf);

打开要传输的文件，将文件内的字符读取到字符缓冲区buf中，然后将buf作为HTTP应答报文的应答体与应答行和应答头进行拼接。需要注意的是，报文的每一行都要以\r\n结尾，应答头与应答体之间也要用\r\n分隔。

2）建立socket文件描述符

// create socket

if ((s = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

perror("create socket failed");

return -1;

}

printf("socket created");

3）将socket文件描述符与监听地址绑定

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = INADDR\_ANY;

server.sin\_port = htons(80);

// bind

if (bind(s,(struct sockaddr \*)&server, sizeof(server)) < 0) {

perror("bind failed");

return -1;

}

printf("bind done");

4）进行监听

// listen

listen(s, 6);

printf("waiting for incoming connections...");

5）接受连接请求，创建子进程以服务连接

while(1)

{

// accept connection from an incoming client

int c = sizeof(struct sockaddr\_in);

if ((cs = accept(s, (struct sockaddr \*)&client, (socklen\_t \*)&c)) < 0) {

perror("accept failed");

return -1;

}

printf("connection accepted");

pid = fork();

if(pid==0)

{

//child

close(s);

int msg\_len = 0;

// receive a message from client

while ((msg\_len = recv(cs, msg, sizeof(msg), 0)) > 0) {

// send the message back to client

printf("%s\n",msg);

char message[4000]={0};

strcat(message,msg);

for(int j = 19; j < strlen(msg);j++)

message[j] = 0;

printf("%s\n",message);

if(strcmp(message,model) == 0)

{

write(cs,OK,strlen(OK));

}

else write(cs,no,strlen(no));

}

if (msg\_len == 0) {

printf("client disconnected");

}

else { // msg\_len < 0

perror("recv failed");

close(cs);

return -1;

}

close(cs);

return 0;

}

else if(pid > 0)

{//parent

close(cs);

}

else

{

printf("fork failed!\n");

exit(1);

}

}

在创建子进程后，在子进程内，首先要执行close（s），因为s是绑定到监听地址和端口的描述符，在建立连接后子进程不再需要继续监听。相应的，cs是接受了来自监听地址和端口的请求后的接受描述符，使用该描述符通信的只有为此专门创建的子进程，父进程不需要与之通信，故父进程内要执行close（cs）。

子进程使用recv接受从客户端发来的HTTP请求报文，从中提取出请求行后判断该请求是否满足HTTP格式，所请求的对象是否在本地存在，若存在就将200 OK的应答行，描述文件长度的应答头和文件内容的应答体组成的应答报文通过write的方式返回给客户端。然后继续等待消息到达。直到下一个到达报文的长度为0，则跳出循环终止连接，结束子进程。

二、客户端

1）建立socket文件描述符

// create socket

sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock == -1) {

printf("create socket failed");

return -1;

}

printf("socket created");

2）申请建立连接

// connect to server

if (connect(sock, (struct sockaddr \*)&server, sizeof(server)) < 0) {

perror("connect failed");

return 1;

}

printf("connected\n");

3）数据传输

char sendData[] = "GET /1.txt HTTP/1.1\r\n\r\n";

char sendData2[] = "GET /2.txt HTTP/1.1\r\n\r\n";

while(1) {

printf("enter message : ");

scanf("%d", &file\_name);

// send some data

if(file\_name == 1)

{

if (send(sock, sendData, strlen(sendData), 0) < 0) {

printf("send failed");

return 1;

}

}

else {if (send(sock, sendData2, strlen(sendData2), 0) < 0) {

printf("send failed");

return 1;

}

}

// receive a reply from the server

int len = recv(sock, server\_reply, 2000, 0);

if (len < 0) {

printf("recv failed");

break;

}

server\_reply[len] = 0;

if(server\_reply[9] == '2' )

printf("HTTP 200 OK\n");

if(server\_reply[9] == '4' )

{

printf("HTTP 404 FILE NOT FOUND\n");

continue;

}

printf("send massage is : %s\n",file\_name == 1?sendData:sendData2);

printf("%s\n",server\_reply);

printf("content length is : ");

char \*find = "Content-Length: ";

char \*s1,\*s2;

int position = 0;

int i = 0;

int length = 0;

while(server\_reply[i] != '\0')

{

s1 = &server\_reply[i];

s2 = find;

while(\*s1 == \*s2 && \*s1 != '\0' && \*s2 != '\0')

{

s1++;

s2++;

}

if(\*s2 == '\0')

break;

i++;

position++;

}

i = position + 16;

while(server\_reply[i]>='0' && server\_reply[i] <= '9')

{

//printf("%c",server\_reply[i]);

length = 10\*length + server\_reply[i] - '0';

i++;

}

printf("%d\n",length);

printf("content :");

s1 = &server\_reply[len - length];

printf("%s\n", s1);

fd = open("1\_receive.txt",O\_RDWR|O\_CREAT|O\_APPEND);

if(fd == -1)

{

printf("create file failed\n");

return 1;

}

write(fd,s1,length);

}

close(sock);

首先准备好要发送的报文内容，然后按照请求行、请求头和请求体的顺序进行拼接。需要注意的是，报文的每一行都要以\r\n结尾，请求头与请求体之间也要用\r\n分隔。

recv接收了发送过来的消息后，首先识别应答头是不是为200 OK，如果是的话，在应答体中寻找Content-length获得文本长度，随后从发送过来的文本末端提取Content-length个字节的长度，然后将此部分数据保存至本地。