

实验报告

Socket 应用编程实验

一、实验内容

使用 Socket API 实现最简单的 HTTP 服务器和 HTTP 客户端。HTTP 服务器侦听 80 端口的 HTTP 请求，支持对 GET 请求的应答。HTTP 客户端需要支持发送 GET 请求，以及接收服务器的回应。服务器还需要支持多线程，支持多个客户端同时请求。

二、实验流程

1. 使用 Socket API 实现符合要求的 HTTP 服务器和 HTTP 客户端。
2. 测试使用客户端请求存在的文件的简单情况。
3. 测试使用客户端请求不存在的文件的情况。
4. 测试使用客户端连续多次获取文件。
5. 测试同时启动多个客户端进程分别获取文件。
6. 使用 `python -m SimpleHTTPServer 80` 和 `wget` 替代服务器和客户端程序，与自己实现的服务器和客户端进行对比验证。

三、实验结果及分析

（一）服务器与客户端设计思路

1、服务器设计思路

服务器首先建立 socket 文件描述符，然后绑定监听地址，最后对 80 端口进行监听。然后是服务器核心部分，循环不断接收连接请求，在收到一个服务器的连接后，接收并解析其 HTTP 请求，然后根据请求内容进行应答。该部分具体代码如下所示。

```

while (1) {
    if ((cs[thread_id] = accept(s, (struct sockaddr *)&(client[thread_id]), (socklen_t *)&c)) < 0) {
        perror("accept failed");
        return -1;
    }
    printf("connection accepted\n");
    fflush(stdout);

    pthread_create(&thread[thread_id], NULL, handle_request, &cs[thread_id]);
    thread_id = (thread_id + 1) % 1024;
    //handle_request(cs);
}

```

其中与一个客户端建立连接后，使用 `pthread_create` 创建一个线程调用 `handle_request()` 处理该线程的请求。而主线程继续执行循环，等待下一个连接的客户端。由此实现服务器的多线程多路并发处理。

处理请求的函数功能为：

首先，接收客户端的 request 报文。

```

// receive a message from client
request_len = recv(cs, request, 2000, 0);
if (request_len < 0) {
    printf("recv failed\n");
    return NULL;
}
request[request_len] = '\0';

```

拿到报文后，检查是否为 GET 请求。

```

char *req_get = strstr(request, "GET ");

```

如果不是 GET，返回 400 Bad Request

```

// other request except GET
sprintf(response, "HTTP/1.1 400 Bad Request\r\nConnection: Close\r\n\r\n");
if (send(cs, response, strlen(response), 0) < 0) {
    printf("send failed\n");
    return NULL;
}
printf("Other Request, rsp_size: %ld\n", strlen(response));

```

如果是 GET 请求，根据 URL 拿到目标文件路径，检查文件是否存在。

```

// open target file
FILE *fp = fopen(file_name, "r");

```

如果不存在，返回 404 File Not Found

```

sprintf(response, "HTTP/1.1 404 File Not Found\r\nConnection: Close\r\n\r\n");
if (send(cs, response, strlen(response), 0) < 0) {
    printf("send failed\n");
    return NULL;
}
printf("File does not exist, rsp_size: %ld\n", strlen(response));

```

如果文件存在，表明请求成功，返回 200 OK，并将文件传送过去。

```

// send the message back to client
sprintf(response, "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nConnection: Close\r\n\r\n");
int rsp_length = strlen(response);
for (i=0; i<rsp_length; i++) {
    write(cs, &response[i], 1);
}
for (i=0; i<file_size; i++) {
    write(cs, &tmp[i], 1);
}
printf("Send Success, rsp_size: %d\n", rsp_length + file_size);

```

2、客户端设计思路

客户端首先从命令行参数中获取 ip、port 号、目标文件路径等信息。然后建立 socket 文件描述符，向服务器发起连接。当连接上服务器，就发送 GET 请求报文。

```

// send get request
printf("send GET request: %s\n", input);
sprintf(request, "GET %s HTTP/1.1\r\nHost: %s:%d\r\nConnection: Close\r\n\r\n", file_path, ip, port);

if (send(sock, request, strlen(request), 0) < 0) {
    printf("send failed\n");
    return 1;
}

```

之后等待接收服务器回复报文。收到回复报文后，对其进行解析，分为三种情况。

```

// process response messages
char *state_ok = strstr(server_reply, "200 OK");
char *state_false = strstr(server_reply, "404 File Not Found");
char *state_bad = strstr(server_reply, "400 Bad Request");

```

对于 404、400 打印报错信息，对于三种情况之外，打印报文信息。

对于正常请求成功的 200 OK 情况，则处理报文主体，写入到本地文件。

```

char *header_end = strstr(server_reply, "\r\n\r\n");
char file_name[20] = "client.";
strcpy(&file_name[strlen(file_name)], file_path + 1);
printf("%s\n", file_name);
FILE *fp = fopen(file_name, "w+");
if (header_end && fp) {
    int body = (header_end - server_reply) + 4;
    server_reply[body-1] = '\0';
    printf("%s", server_reply);
    fprintf(fp, "%s", &server_reply[body]);
    fflush(stdout);
    fclose(fp);
}
else{
    printf("%s", server_reply);
}

```

通过 “\r\n\r\n” 的 header 结束标志找到报文的主体部分。将 header 信息打印到屏幕，主体部分写入本地文件。

（二）测试功能

1、客户端请求存在的文件

使用客户端对存在的文件 “test.html” 进行请求。

```

root@joker-linux:/mnt/hgfs/share/03-socket# ./http-client http://10.0.0.1:80/test.html
ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test.html
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test.html
Server Reply: (7445)
client.test.html
HTTP/1.1 200 OK
Content-Type: text/html
Connection: Close

```

其中 7-9 行内容为客户端发送的请求报文。可以看出其正确解析了 ip、port 号和目标文件名。

后面为服务器返回内容的解析。回应报文显示为 200 OK，表明请求成功，然后将报文主体的内容打印至本地文件：client.test.html。

之后使用 diff 验证文件。

```

root@joker-linux:/mnt/hgfs/share/03-socket# diff test.html client.test.html
root@joker-linux:/mnt/hgfs/share/03-socket# █

```

可以看出，保存的文件与服务器原始文件相同，传输成功。

再看这一过程中服务器的打印信息。

```
root@joker-linux:/mnt/hgfs/share/03-socket# ./http-server 80
socket created
bind done
waiting for incoming connections... port: 80
connection accepted

request: (65)
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test.html
Send Success, rsp_size: 7445
█
```

可以看到 8-10 行内容为服务器收到的请求报文。服务器根据报文查找到本地文件“test.html”存在，于是成功发送回应报文。

2、客户端请求不存在的文件

使用客户端对不存在的文件“nofile.html”进行请求。

```
root@joker-linux:/mnt/hgfs/share/03-socket# ./http-client http://10.0.0.1:80/nofile.html
ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/nofile.html
GET /nofile.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.nofile.html
Server Reply: (50)
HTTP/1.1 404 File Not Found
Connection: Close

404 File Not Found : The target file does not exist.
```

回应报文显示为 404 File Not Found，表明请求失败，打印报错信息。

再看这一过程中服务器的打印信息。

```
root@joker-linux:/mnt/hgfs/share/03-socket# ./http-server 80
socket created
bind done
waiting for incoming connections... port: 80
connection accepted

request: (67)
GET /nofile.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

nofile.html
File does not exist, rsp_size: 50
█
```

服务器解析目标文件为 onfile.html，该文件实际不存在，于是发送 404 回应报文。

可以看出对于不存在文件的情况，客户端和服务端可以正常工作。

3、客户端连续多次获取文件

使用客户端连续多次获取文件“test.html”。

```
root@joker-linux:/mnt/hgfs/share/03-socket# ./http-server 80
socket created
bind done
waiting for incoming connections... port: 80
connection accepted

request: (65)
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test.html
Send Success, rsp_size: 7445
connection accepted

request: (65)
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test.html
Send Success, rsp_size: 7445
connection accepted

request: (65)
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test.html
Send Success, rsp_size: 7445
connection accepted

request: (65)
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

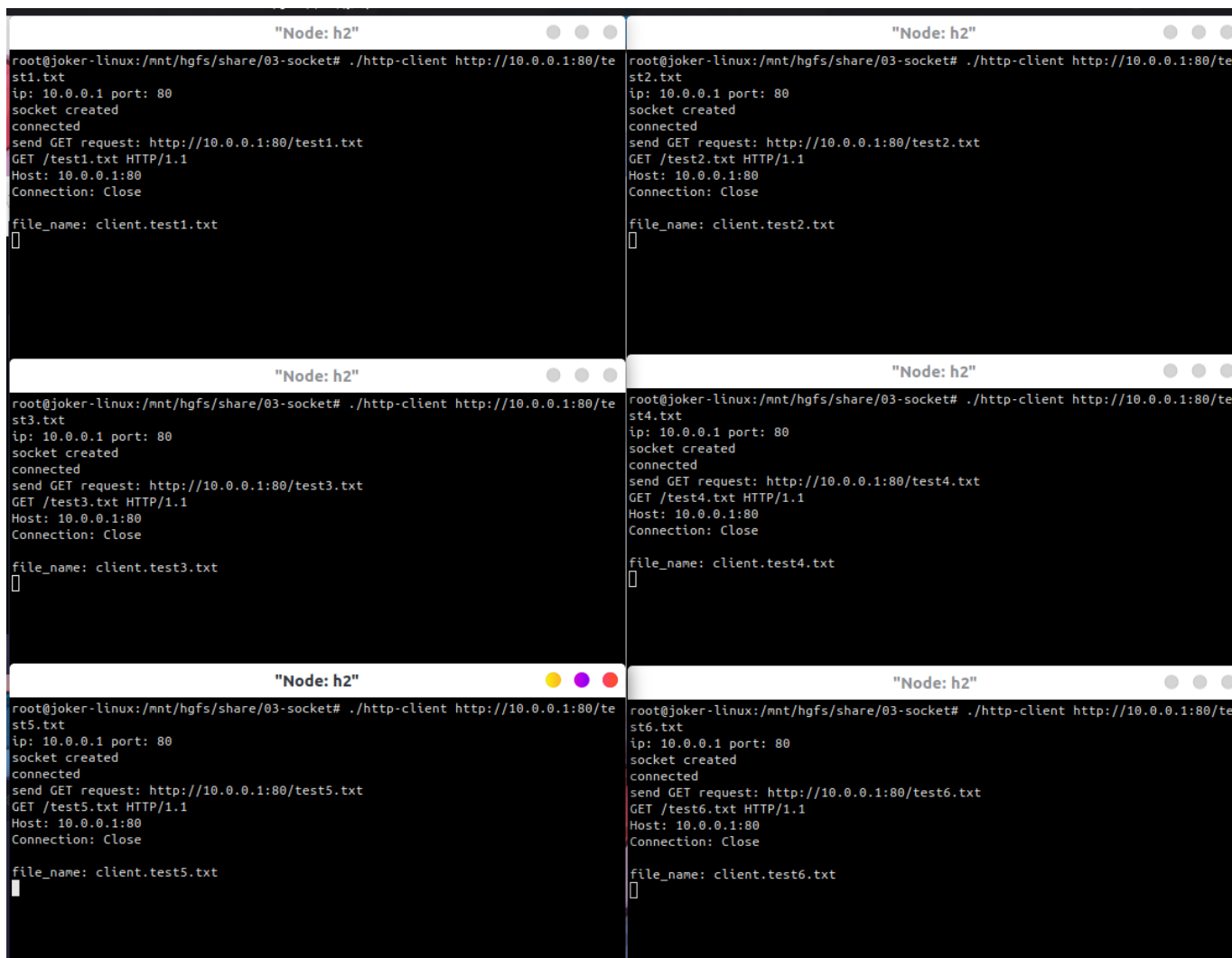
test.html
Send Success, rsp_size: 7445
```

根据服务器工作信息看出，每次请求都成功接收，并正确处理。之后对每次传输的文件都用 diff 进行验证，也全部验证正确。

可以看出对于连续多次获取文件的情况，服务器可以正常工作。

4、同时启动多个客户端进程分别获取文件

同时启动多个客户端分别获取文件。为了能看出同时获取文件的效果，我们将文件大小增加到 10MB 左右，传输大概需要 10 秒。由此可以观察，服务器能否同时接收和处理多个请求，并在同时使用多个线程并行进行文件传输。



The image displays six terminal windows arranged in a 3x2 grid, each titled '"Node: h2"'. Each window shows the execution of an HTTP client command to fetch a specific file from a server at 10.0.0.1:80. The commands and their outputs are as follows:

- Top Left:** `./http-client http://10.0.0.1:80/test1.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test1.txt
GET /test1.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test1.txt`
- Top Right:** `./http-client http://10.0.0.1:80/test2.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test2.txt
GET /test2.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test2.txt`
- Middle Left:** `./http-client http://10.0.0.1:80/test3.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test3.txt
GET /test3.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test3.txt`
- Middle Right:** `./http-client http://10.0.0.1:80/test4.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test4.txt
GET /test4.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test4.txt`
- Bottom Left:** `./http-client http://10.0.0.1:80/test5.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test5.txt
GET /test5.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test5.txt`
- Bottom Right:** `./http-client http://10.0.0.1:80/test6.txt`
Output: `ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test6.txt
GET /test6.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test6.txt`

这是多个客户端同时发起获取文件的请求，他们都处于等待回复的状态。

此时服务器状态为：

```
request: (65)
GET /test5.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test5.txt
connection accepted

request: (65)
GET /test6.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test6.txt
connection accepted

request: (65)
GET /test1.txt HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

test1.txt
█
```

可以看出服务器同时建立了多个连接，接受了多个 GET 请求。

接下来观察到各个客户端相继收到回应报文，完成传输。

而服务器打印显示：

```
Send Success, rsp_size: 13000063
Send Success, rsp_size: 13000063
Send Success, rsp_size: 13000063
Send Success, rsp_size: 13000063
Send Success, rsp_size: 13000063
Send Success, rsp_size: 13000063
```

这说明 6 个连接的回复报文同时发出。

如果服务器是串行接收请求，那打印效果应该是一个 request GET 后面一个 Send Success。而现在是多个 request GET 一起打印，最后多个 Send Success 一起打印。由此看出服务器确实使用了多个线程分别处理请求。

综上可以看出服务器确实支持多线程并发工作。

5、使用 SimpleHTTPServer 和 wget 对比验证

使用 `python -m SimpleHTTPServer 80` 和 `wget` 替代服务器和客户端程序，进行对比验证。

首先使用 `python -m SimpleHTTPServer 80` 作为服务器，使用自己编写的客户端程序发送请求。


```

root@joker-linux:/mnt/hgfs/share/03-socket# ./http-client http://10.0.0.1:80
st.html
ip: 10.0.0.1 port: 80
socket created
connected
send GET request: http://10.0.0.1:80/test.html
GET /test.html HTTP/1.1
Host: 10.0.0.1:80
Connection: Close

file_name: client.test.html
Server Reply: (7569)
client.test.html
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/2.7.18
Date: Fri, 26 Mar 2021 13:35:46 GMT
Content-type: text/html
Content-Length: 7382
Last-Modified: Thu, 25 Mar 2021 09:47:55 GMT

```

可以看出，对于 SimpleHTTPServer 服务器，自己编写的客户端可以正确完成发送请求报文和解析回应报文的工作。同时，我们可以看到 SimpleHTTPServer 服务器的回应报文内容更多，它加入了时间、文件长度、文件最后修改时间等信息。

```

root@joker-linux:/mnt/hgfs/share/03-socket# python2 -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.2 - - [26/Mar/2021 21:35:46] "GET /test.html HTTP/1.1" 200 -

```

而 SimpleHTTPServer 服务器本身打印的信息就比较少了。它只打印了请求报文的首句、回应状态代码 200 以及处理请求的时间。

然后我们再使用自己编写的 http-server 作为服务器，用 wget 发送请求。

```

root@joker-linux:/mnt/hgfs/share/03-socket# wget http://10.0.0.1:80/test.html
--2021-03-26 21:46:08-- http://10.0.0.1/test.html
正在连接 10.0.0.1:80... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度： 未指定 [text/html]
正在保存至: "test.html.1"

test.html.1          [ <=>          ]  7.21K  --.-KB/s    用时 0.004s
2021-03-26 21:46:08 (1.86 MB/s) - "test.html.1" 已保存 [7382]

```

wget 与自己编写的客户端打印信息差别不大，wget 也给出了建立连接、发出请求、回应状态 200 OK、写入文件等信息。区别在于 wget 提供了一个传输文件的进度条，可以让用户看到传输进度。

```

root@joker-linux:/mnt/hgfs/share/03-socket# ./http-server 80
socket created
bind done
waiting for incoming connections... port: 80
connection accepted

request: (144)
GET /test.html HTTP/1.1
User-Agent: Wget/1.20.3 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: 10.0.0.1
Connection: Keep-Alive

test.html
Send Success, rsp_size: 7445
[]

```

再看 http-server 上的信息，可以看出 wget 发送的请求报文内容更多。它增加了 wget 自身的信息、accept 信息等。http-server 与 SimpleHTTPServer 对比，我们的服务器打印信息更多，能更清晰展示服务器处理过程，便于调试。

另外，对于文件不存在的情况的回应报文，我们的 http-server 也正确处理，可以被 wget 识别。

```

root@joker-linux:/mnt/hgfs/share/03-socket# wget http://10.0.0.1:80/te.html
--2021-03-26 21:52:44-- http://10.0.0.1/te.html
正在连接 10.0.0.1:80... 已连接。
已发出 HTTP 请求，正在等待回应... 404 File Not Found
2021-03-26 21:52:44 错误 404: File Not Found。

```

6、使用客户端向 baidu 发送请求验证其功能的正确性

这部分属于一个实验性质的测试，既然我们客户端可以向内网服务器发送请求，自然也应该可以向互联网服务器发送请求。我们尝试用客户端下载 baidu 页面，来验证其 HTTP 报文功能的正确性。

不使用 mininet 环境，直接使用 http-client 客户端对 baidu (ip: 220.181.38.149) 发送请求。

```

joker@joker-linux:/mnt/hgfs/share/03-socket$ ./http-client http://220.181.38.149:80/
ip: 220.181.38.149 port: 80
socket created
connected
send GET request: http://220.181.38.149:80/
GET / HTTP/1.1
Host: 220.181.38.149:80
Connection: Close

file_name: client.
Server Reply: (15570)
client.
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: no-cache
Content-Length: 14615
Content-Type: text/html
Date: Fri, 26 Mar 2021 14:04:26 GMT
P3p: CP=" OTI DSP COR IVA OUR IND COM "
P3p: CP=" OTI DSP COR IVA OUR IND COM "
Pragma: no-cache
Server: BWS/1.1

```

可以看到客户端发送的请求被 `baidu` 服务器接收了，并成功下载了 `baidu` 页面。`Baidu` 服务器发回的报文的 `header` 内容更多，有 `cache`、`date`、`p3p` 等，未截图的部分还有 `cookie`、`trace` 等等。可以看出实际使用的 `HTTP` 报文内容更多，条目更为复杂。

由此，我们也验证了客户端的 `HTTP` 报文的正确性。

四、实验总结

通过本次实验，我对 `Socket API` 有了一定的了解。`Socket API` 对上层提供统一的调用接口，提供最基本的网络通信功能。通过实际编写一个简单的 `HTTP` 服务器与客户端，我对于主从两方建立 `Socket` 描述符、建立连接等内容都有了不少了解。对于客户端、服务器之间的交互机制也有了一定的认识。此外我也学到了 `HTTP` 协议的内容，主要是对 `HTTP` 报文的结构有了不少了解，这让我对网络文件传输有了更深的理解。