

计算机网络实验 3 报告

学号 2017K8009929059

姓名 於修远

Socket 应用编程实验

一、实验内容

- 在课程所提供的 server 端与 client 端的 Socket 编程代码基础上，修改编写实现 HTTP GET 方法进行文件传输的 server 端与 client 端应用。
- 使用脚本`topo.py`建立网络实验环境，测试服务器与用户端可实现连续请求文件、多客户端连接、请求不存在文件。
- 通过 Python 内置的 SimpleHTTPServer 模块和 wget 指令分别测试用户端和服务器的工作情况。

二、实验流程

(一) Client 编写

1、连接的建立

首先，用户端程序通过输入的 URL 建立向对端的连接，端口号为宏定义中设定的 80。这一 URL 会在后续的报文中被填入 HOST 字段中，本实验中固定为"10.0.0.1"。

建立连接阶段的代码基本沿用课程所提供的示例，即先创建 Socket 描述符，再申请进行连接。之后的所有断开连接以外的操作用一个`while(1)`循环包括，以实现长连接，通过 ctrl+c 中止。

2、HTTP 请求的组装与发送

大循环体的每一次执行，都以用户输入需要下载的文件名开始，这一字段被储存在 filename 中，随后，通过 sprintf 函数实现 HTTP 请求的组装。

```
// prepare http packet
memset(buf, 0, sizeof(buf));
sprintf(buf, format: "%s/%s%s\r\n\r\n", request_head1, filename, request_head2, url);
```

其中的 request_head 字段均为常量。本设计中采用 HTTP-1.1 协议，同时添加 HOST 请求头。HTTP-1.1 协议默认开启请求头部的长连接 keep-alive 选项，方便报文组装生成。

```
char request_head1[] = "GET ";
char request_head2[] = " HTTP/1.1\r\nHost: ";
```

随后使用 `send` 函数发送 HTTP 请求，并使用 `recv` 函数阻塞式接收服务器反馈的消息。当 `recv` 返回值为 0 时，意味着服务器断开连接，此时用户端程序也跳出请求循环，关闭连接。

```
// receive a reply from the server
int len = 0;
memset(buf, 0, sizeof(buf));
while ((len = recv(sock, buf, BUFSIZE, flags: 0)) != 0) {...}
// break
if (len == 0) break;
```

3、HTTP 响应解析

对于接收到的 HTTP 响应，第一步是解析状态码，方法是用 `sscanf` 函数正则匹配获取响应报文的第一第二个空格间的字符串。由于仅要求实现 404 和 200 两种状态码，所以这里简单地将状态码与 404 对比作为响应结果。对于 404 的响应，仅打印相关信息。

```
char state[4];
sscanf(buf, format: "%*[^ ] %[^ ]", state);
if (strcmp(state, "404") != 0) {...}
else {
    printf( format: "404 File Not Found.\n");
    break;
}
```

对于反馈为 200 的有效报文，首先通过匹配 `\r\n\r\n` 确定报文中响应数据部分的起始位置，然后将从该位置开始的有效信息写入预先打开的文件中。为避免与服务器端文件混淆，本程序中 `client` 的默认下载路径为 `./download/`。完成后关闭文件描述符即可。

```
int start = 0;
while (start < len) {
    if (buf[start] == '\r' && buf[start + 2] == '\r' && buf[start + 1] == '\n' && buf[start + 3] == '\n') {
        start += 4;
        break;
    }
    start++;
}
FILE *fp = fopen(filename, modes: "a+");
unsigned long wret = fwrite(&buf[start], sizeof(char), n: len - start, fp);
```

(二) Pthread-Server 编写

1、端口监听

服务器端代码的初始化部分同样基本复用课程所提供代码，即创建并绑定描述符、进行监听。增加了一项 `socket` 描述符选项的设置，打开地址复用选项，便于后续的多线程对同一端口地址进行使用。

```

int yes = 1;
if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) < 0) {
    perror(s: "Setsockopt error");
    return -1;
}

```

2、多客户端连接响应的实现

监听完成后，程序进入一个 `while(1)` 循环，每个循环都会在申请一个新的连接套接字空间后进入 `accept` 函数的阻塞中。当接收到来自客户端的 `connect` 请求，`accept` 函数会响应并将对应连接套接字存入连接套接字 `cs` 中，并使用 `pthread_create` 函数创建一个新线程并传入连接套接字。此后，这一连接的所有信息收发都由新创建的子进程进行管理，管程则重新进入 `accept` 的阻塞态，等待下一连接进入。

```

while (1) {
    pthread_t thread;
    cs = (int*)malloc(sizeof(int));

    // accept connection from an incoming client
    int c = sizeof(struct sockaddr_in);
    if ((*cs = accept(s, (struct sockaddr *)&client, (socklen_t *)&c)) < 0) {...}
    printf(format: "Connection from %s accepted.\n", inet_ntoa(client.sin_addr));
    if (pthread_create(&thread, attr: NULL, routine, cs) != 0) {...}
}

```

3、HTTP 通讯实现

在服务器端，采用一个 `while(1)` 循环实现与客户端的长连接。当 `recv` 返回值非正时，意味着连接错误或客户端连接断开，此时服务器也做出相应调整：关闭连接，并释放独占的一个套接字内存空间。该线程的工作也就此结束。

```

while (1) {
    // receive a message from client
    if ((len = recv(fd, buf, sizeof(buf), flags: 0)) > 0) {...}
    else {
        if (len == 0) {
            printf(format: "Client disconnected\n");
        } else { // len < 0
            perror(s: "Recv failed");
        }
        close(fd);
        free(cs);
        pthread_exit(retval: NULL);
    }
}

```

服务器端的 HTTP 请求解析做了比较大程度的精简，直接正则匹配请求行中的文件名部分。由于本次实验的聚焦点为 `socket` 连接和 HTTP 传输，所以没有添加路径支持，只查找服务器所在目录下的文件来反馈。若当前目录没有请求的文件，则直接装载并发送预设的 404 报文。

```

sscanf(buf, format: "%*s /%[^ ]", filename);
// send the file back to client
FILE *fp = fopen(filename, modes: "r");
if (fp) {...} else {
    printf( format: "File %s not found.\n", filename);
    strcpy(buf, fail);
    if (send(fd, buf, strlen(buf), flags: 0) < 0) {
        printf( format: "Send failed.\n");
    }
    memset(buf, c: 0, sizeof(buf));
}
}

```

正文发送的过程需要引入额外的一个数据缓冲区，用于统计响应数据部分的长度后，与预设的响应行等组装，并写入响应报文头部的 Content-Length 字段值。响应数据的长度上限为读出缓冲区的长度，这里预设为 2000 字节。对于更大文件则需要多次发送报文。

```

while ((rret = fread(readbuf, sizeof(char), CONTENT_SIZE, fp)) > 0) {
    sprintf(buf, format: "%s%lu\r\n\r\n%s", ok, rret, readbuf);
    if (send(fd, buf, strlen(buf), flags: 0) < 0) {...}
    memset(buf, c: 0, sizeof(buf));
}
fclose(fp);

```

(三) Select-Server 编写

1、设计概述

本次实验要求中虽然没有要求编写采用 IO 端口复用的服务器，但由于对这种实现方式比较感兴趣，所以也通过自行查找资料，编写了相应的代码。通过测试，证明这一 Select-Server 可以实现与 Pthread-Server 相同的功能。

具体实现上，端口监听和 HTTP 通讯的实现都与多线程方式的实现基本相同，包括对 socket 描述符的选项设置，无限循环实现长连接和 HTTP 请求解析、响应组装，最主要的区别在于对多客户端响应的处理。对于 select 方式，需要维护一个描述符集合，通过检查集合中描述符的读写变化情况，选出需要分配 IO 端口的连接进行处理。

2、描述符集合的维护

每一轮循环开始时，首先清空维护中的描述符集合，手动添加监听描述符后遍历描述符数组，筛选出其中非零的（即已建立了连接的）描述符。本次循环中，select 所需要检查的描述符就在这一范围内。本实现中设定了 select 的超时为 30s，即每阻塞 30s 重置一次循环，后续可根据需要修改超时后的操作，添加功能。

```

FD_ZERO( fdsetp: &fds);
FD_SET(s, fdsetp: &fds);

for (int i = 0; i < MAXC; i++) {
    if (fd[i] != 0) FD_SET( fd: fd[i], fdsetp: &fds);
}

timeout.tv_sec = 30;
timeout.tv_usec = 0;
int ready = select( nfds: maxs + 1, &fds, writefds: NULL, exceptfds: NULL, &timeout);

```

对于 select 为正的情况，即存在发生了读写情况变动的连接。通过遍历及 `FD_ISSET` 函数的判断，找出这些连接描述符，并进行 HTTP 报文的交互；而对于监听描述符，则在其发生变动时接收新来的连接，并赋给一个空闲的连接描述符。

```

for (int i = 0; i < MAXC; i++) {
    if (FD_ISSET( fd: fd[i], fdsetp: &fds)) {...} // manage selected fd
}
if (FD_ISSET(s, fdsetp: &fds)) {...} // accept a new connection

```

具体实现中还包括连接数量的维护等工程细节，这里不赘述，大体思路即维护连接数不超过预设的描述符数组的容量（这里设定为 8），超过限制时直接拒绝接入。

三、结果分析

（一）Pthread-Server 测试结果

如下图所示，测试了同时使用两个服务器连接 Pthread-Server 的情况，并在最后使用 wget 代替自己编写的 Client。可以看到服务器端正常响应所有请求，检查 download/目录确认文件内容传输正确。

```

example
thread-server.c
"Node:h1"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./pthread-server
Socket created
Bind done
Waiting for incoming connections...
Connection from 10.0.0.2 accepted.
Begin to transfer 111.txt to the client.
Transfer finished.
Connection from 10.0.0.2 accepted.
Begin to transfer 222.txt to the client.
Transfer finished.
File 333.txt not found.
Client disconnected
Connection from 10.0.0.2 accepted.
Begin to transfer 444.txt to the client.
Transfer finished.
Client disconnected
[]

"Node:h2"
root@cod-VirtualBox:~/workspace/ucas_n
URL to connect : 10.0.0.1
Socket created
Connected
Filename to download : 111.txt
200 OK!
File received!
Filename to download : 333.txt
404 File Not Found.
Filename to download : []

"Node:h2"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./client
URL to connect : 10.0.0.1
Socket created
Connected
Filename to download : 222.txt
200 OK!
File received!
Filename to download : ^C
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# wget 10.0.0.1:80/444.t
xt
--2020-04-23 22:49:40-- http://10.0.0.1/444.txt
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 111
Saving to: '444.txt.1'

444.txt.1      100%[=====] 111 --.-KB/s  in 0s

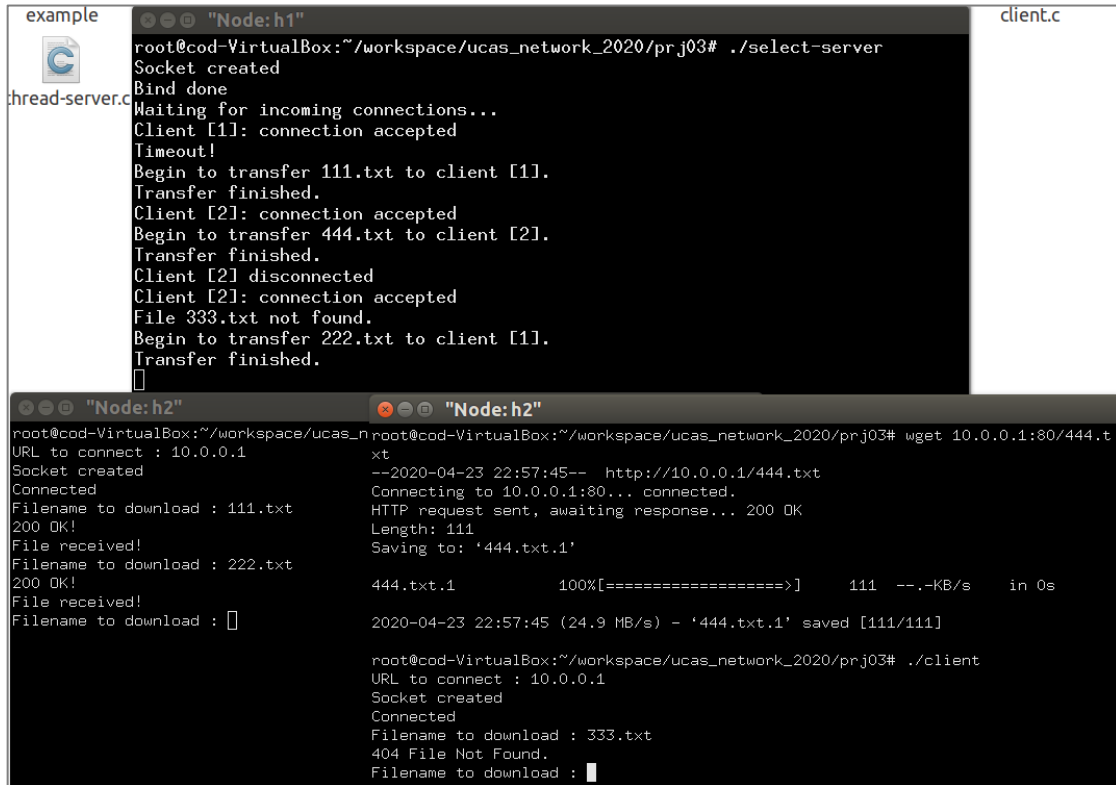
2020-04-23 22:49:40 (29.4 MB/s) - '444.txt.1' saved [111/111]

root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03#

```

(二) Select-Server 测试结果

如下图所示，测试了同时使用两个服务器连接 Select-Server 的情况，并在最后使用 wget 代替自己编写的 Client。可以看到服务器端正常响应所有请求，检查 download/ 目录确认文件内容传输正确。



```
example
thread-server.c

"Node:h1"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./select-server
Socket created
Bind done
Waiting for incoming connections...
Client [1]: connection accepted
Timeout!
Begin to transfer 111.txt to client [1].
Transfer finished.
Client [2]: connection accepted
Begin to transfer 444.txt to client [2].
Transfer finished.
Client [2] disconnected
Client [2]: connection accepted
File 333.txt not found.
Begin to transfer 222.txt to client [1].
Transfer finished.

"Node:h2"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# wget 10.0.0.1:80/444.txt
--2020-04-23 22:57:45-- http://10.0.0.1/444.txt
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 111
Saving to: '444.txt.1'

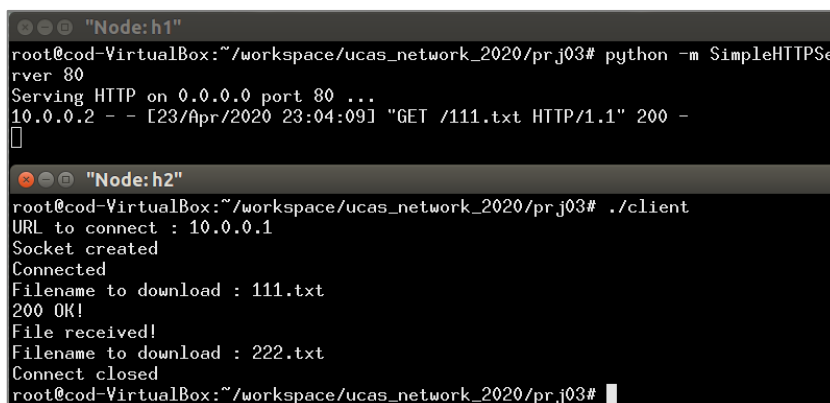
444.txt.1      100%[=====] 111  --.-KB/s  in 0s

2020-04-23 22:57:45 (24.9 MB/s) - '444.txt.1' saved [111/111]

root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./client
URL to connect : 10.0.0.1
Socket created
Connected
Filename to download : 111.txt
200 OK!
File received!
Filename to download : 222.txt
200 OK!
File received!
Filename to download : 
404 File Not Found.
Filename to download :
```

(三) Client 测试结果及分析

如下图所示，通过`python -m SimpleHTTPServer`命令创建服务器后，用 Client 进行连接，会发现第一次传输成功，之后连接直接关闭。



```
"Node:h1"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.2 - - [23/Apr/2020 23:04:09] "GET /111.txt HTTP/1.1" 200 -

"Node:h2"
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./client
URL to connect : 10.0.0.1
Socket created
Connected
Filename to download : 111.txt
200 OK!
File received!
Filename to download : 222.txt
Connect closed
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03#
```

尝试打印出服务器响应报文，发现响应行显示服务器采用 HTTP-1.0，而这默认连接不是长连接。通过查询资料，发现可以通过请求头的`Connection: Keep-Alive`选项启用 HTTP-1.0 的长连接功能，但实际使用后服务器依然会直接中断连接，响应报文与之前没有区别（本应增加响应头中的`Connection: Keep-Alive`作

但实际测试过程中发现与 Client 连接时，第一次响应的报文总会分离成为两段，这一问题暂时还未解决。

```
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# python simpleserver.py
```

Serving at port 80
10.0.0.2 - - [23/Apr/2020 23:53:58] "GET /111.txt HTTP/1.1" 200 -
10.0.0.2 - - [23/Apr/2020 23:54:02] "GET /222.txt HTTP/1.1" 200 -

```
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj03# ./client
```

URL to connect : 10.0.0.1
Socket created
Connected
Filename to download : 111.txt
HTTP/1.1 200 OK

200 OK!
File received!
Filename to download : 222.txt
Server: SimpleHTTP/0.6 Python/2.7.12
Date: Thu, 23 Apr 2020 15:53:58 GMT
Content-type: text/plain
Content-Length: 120
Last-Modified: Tue, 21 Apr 2020 13:57:43 GMT

aaaaaaaaaaaaaaabbbbbbbbbbbbbbccccccccc
ccccccccccccccccdddddrrrrrrrrrrrrrrrrrrr
9999999999999999777777777777777777777777

SimpleHTTP/0.6 OK!
File received!
Filename to download :

（四）分析和总结

在本次实验中，我尝试用两种不同的方式实现了处理多客户端连接的服务器。相较而言，多线程方法的设计更简单些，而且通过查询资料，我也发现 `select` 方法是 IO 复用方法中效率比较差的一种。但 IO 复用的好处在于节省内存，因为观察多线程程序会发现，每个线程都需要申请一块独立的 `buffer` 作为数据收发的缓冲区；而如果共享这一缓冲区，则又需要一个锁机制来调控，`select` 相当于手动实现了一种效率不太高的资源分配方法。同时，IO 复用较之多线程也可以省下很多线程维护的系统开销。

在 **Server-Client** 交互的设计过程中，我更深入地了解 **HTTP** 的传输机制：**Client** 主动建立连接、主动结束连接，服务器端仅负责响应和维护连接。通过本次实验，我对服务器和用户端的传输实现有了更进一步的认识。