计算机网络实验 12 报告

学号　2017K8009929059
姓名　於修远

网络传输机制实验二

一、实验内容

- 补全实验代码，添加` tcp_sock `的读写函数，并调整相关代码，实现 TCP 接收的数据缓存机制。
- 运行给定网络拓扑，用两个节点分别作为服务器节点和客户端节点，数据缓存的正确性。
- 修改` tcp_apps.c `中的相关代码，实现文件收发，并检验其正确性。

二、实验流程

（一）TCP 数据发送

TCP 数据发送的第一步是确定发送数据的长度。考虑到缓冲区的数据可能无法一次性发送完毕，所以添加了一个 while 循环来实现对输入 buf 的检查。在` tcp_sock_write `函数中仅需要将恰当长度的数据拷贝进入数据包空间的对应位置即可，各头部信息会在后续的函数调用中被逐层添加。发送结束后记录已发送长度，并将自身挂起进入已发送队列，直至接收到来自对端的 ACK 信息才会被唤醒，从而防止发送速度过快引起错误。

```
int sent = 0;
while (sent < len) {
    int valid_len = min(len, y: strlen(buf)) - sent;
    int data_len = min(MSS, valid_len);
    int pkt_len = TCP_PKT_LEN(data_len);
    char *packet = (char *) malloc(pkt_len);
    memcpy( dest: packet + TCP_BASE_ALL_HDR_SIZE, src: buf + sent, data_len);
    tcp_send_packet(tsk, packet, pkt_len);
    sent += data_len;
    sleep_on(tsk->wait_send);
}
return sent;
```

（二）TCP 数据接收

由于涉及环形 buffer 的读取，所以` tcp_sock_read `函数需要在互斥锁内完成。首先循环检测接收缓存是否为空，为空时释放锁资源并挂起，等待数据到达后重新申请锁资源。数据读取直接调用框架内函数即可。

```
pthread_mutex_lock(&tsk->rcv_buf->lock);
while (ring_buffer_empty(tsk->rcv_buf)) {
    pthread_mutex_unlock(&tsk->rcv_buf->lock);
    sleep_on(tsk->wait_recv);
    pthread_mutex_lock(&tsk->rcv_buf->lock);
}
int res = read_ring_buffer(tsk->rcv_buf, buf, len);
pthread_mutex_unlock(&tsk->rcv_buf->lock);
return res;
```

（三）TCP 状态机调整

1、数据 ACK 的处理

如下图所示，在接收到 ACK 信号时的处理中增加了状态为 ESTABLISHED 的情况，具体行为为释放发送等待区的线程。这一设计一方面便于流量控制，另一方面也可以防止数据发送方过早地发出 FIN 信号，使对端接收不完整。

```
case TCP_ACK:
    switch (tsk->state) {
        case TCP_SYN_RECV:
            tcp_sock_accept_enqueue(
            wake_up(tsk->parent->wai
            tcp_set_state(tsk,  state:
            break;
        case TCP_ESTABLISHED:
            wake_up(tsk->wait_send);
            break;
```

2、数据 PSH 的处理

如下图所示，针对` tcp_send_packet `函数发送的 PSH 和 ACK 位为 1 的数据包，申请锁资源后写入接收缓存区，随后唤醒被阻塞的读取进程，并向对端回复 ACK 确认消息。此处，最后的唤醒等待队列操作是为了防止对端未发送 ACK 信号而引起死锁：以 python 脚本为服务器端时就会遇到这种情况。

```
case (TCP_PSH | TCP_ACK):
    pthread_mutex_lock(&tsk->rcv_buf->lock);
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    pthread_mutex_unlock(&tsk->rcv_buf->lock);
    if (tsk->wait_recv->sleep) {
        wake_up(tsk->wait_recv);
    }
    tcp_send_control_packet(tsk, TCP_ACK);
    if (tsk->wait_send->sleep) {
        wake_up(tsk->wait_send);
    }
    break;
```

（四）文件收发实现

修改后的源文件通过宏定义控制编译得到的文件是否支持文件传输。其中，文件传输的输入默认为` client-input.dat `，接收方将接收到的每个包在终端回显，并写入` server-output.dat `文件，理论上支持任意大小的文件传输，文件读取缓存设置为 1000 字节。由于上述循环发送机制的存在，理论上可为任意大小。

```
#define FILE_TRANSFER

#ifdef FILE_TRANSFER
#define INPUT_NAME "client-input.dat"
#define OUTPUT_NAME "server-output.dat"
#endif
```

发送方函数的核心内容如下图所示。每次从文件中读出不超过临时缓冲区` wbuf `大小的数据，并调用` tcp_sock_write `函数作为待发送数据，直至将文件读取完毕。

```
while ((wlen = fread(wbuf, sizeof(char), CONTENT_SIZE, fp)) > 0) {
    printf( format: "[Send] %s\n", wbuf);
    if (tcp_sock_write(tsk, wbuf, wlen) < 0) {
        printf( format: "Unexpected ERROR!!!!!!\n");
        break;
    }
    // usleep(1000);
    memset(wbuf,  c: 0, sizeof(wbuf));
    if (feof(fp)) break;
}
```

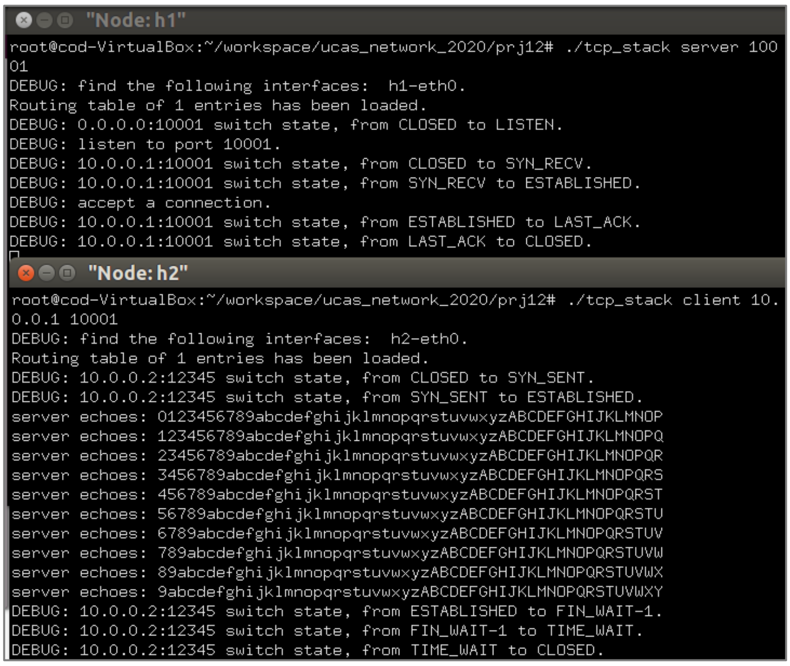
是接收方函数的核心内容如下图所示。在传输完成或出现错误前循环读取接收缓冲区内的数据，同样将其缓存到文件缓冲区` rbuf `中。此后，再将缓冲区数据写入文件即可。这里调用` fflush `函数可以保证数据被及时写入数据，防止传输错误等情况影响下写入缺失的情况。

```
while (1) {
    memset(rbuf,  c: 0, sizeof(rbuf));
    rlen = tcp_sock_read(csk, rbuf, CONTENT_SIZE);
    if (rlen == 0) {
        log(DEBUG, "tcp_sock_read return 0, finish transmission.");
        break;
    }
    else if (rlen > 0) {
        rbuf[rlen] = '\0';
        printf( format: "[Recv] %s\n", rbuf);
        wlen = fwrite(rbuf, sizeof(char), rlen, fp);
        fflush(fp);
        if (wlen < rlen) {
            printf( format: "File writing fail!\n");
            break;
        }
    }
```

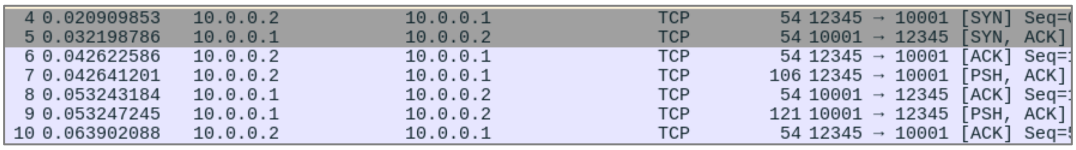
三、结果分析

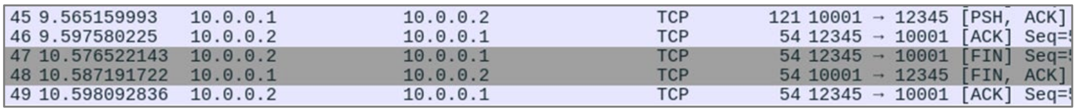
（一）字符串传输

如下图所示，执行` tcp_topo.py `脚本后，将 h1 节点作为 server 端，将 h2 节点作为 client 端，建立 tcp 连接。两节点间的交互结果如下图所示，可见连接正常。

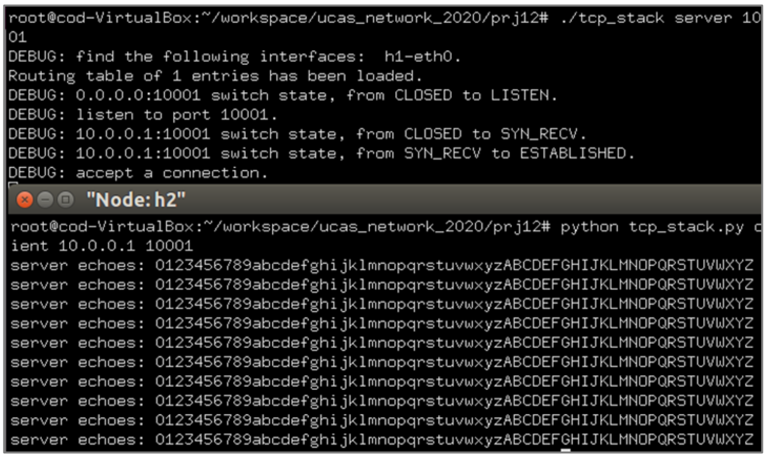


通过 wireshark 查看节点 h2 的收发包情况，可以看到 TCP 连接正常建立到关闭的过程。

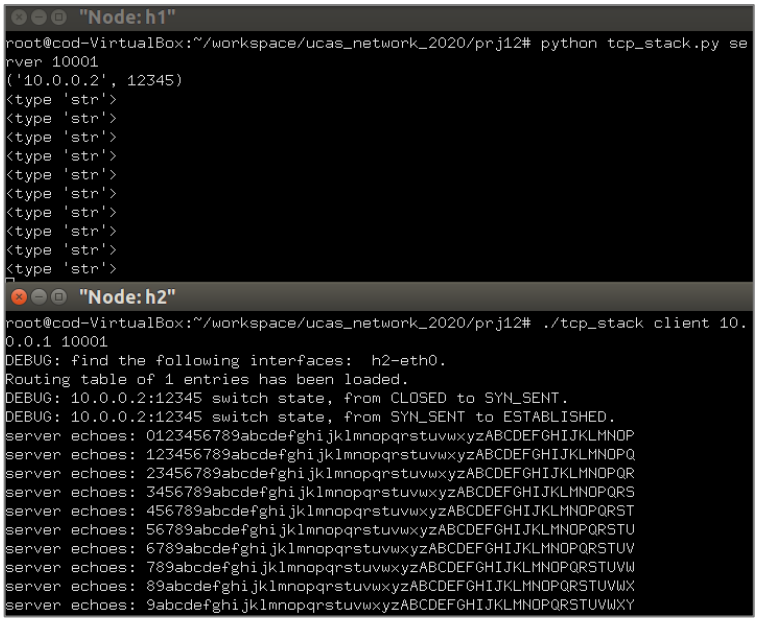




如下图所示，以` tcp_stack.py `脚本替代用户端后，连接依然正常。

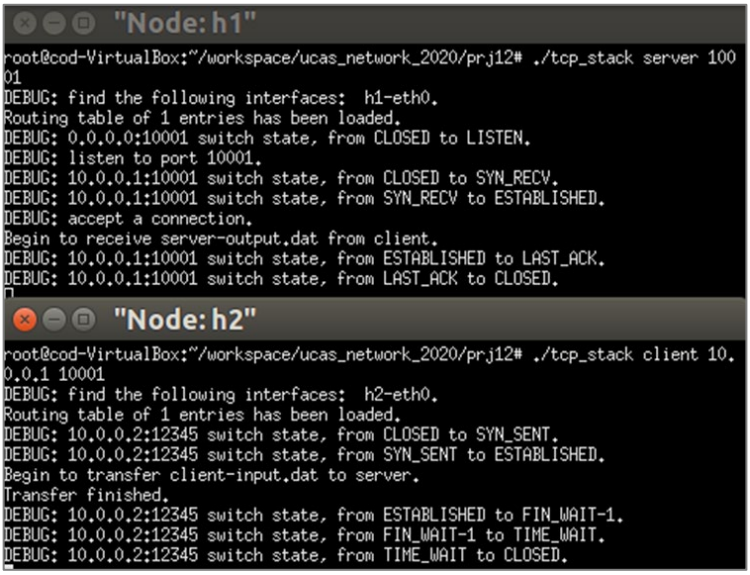


如下图所示，以` tcp_stack.py `脚本替代服务器端后，连接依然正常。

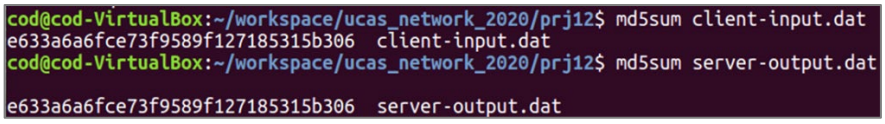


（二）文件传输

如下图所示，执行` tcp_topo.py `脚本后，将 h1 节点作为 server 端，将 h2 节点作为 client 端，建立 tcp 连接。两节点间的交互结果如下图所示，可见传输过程正常。

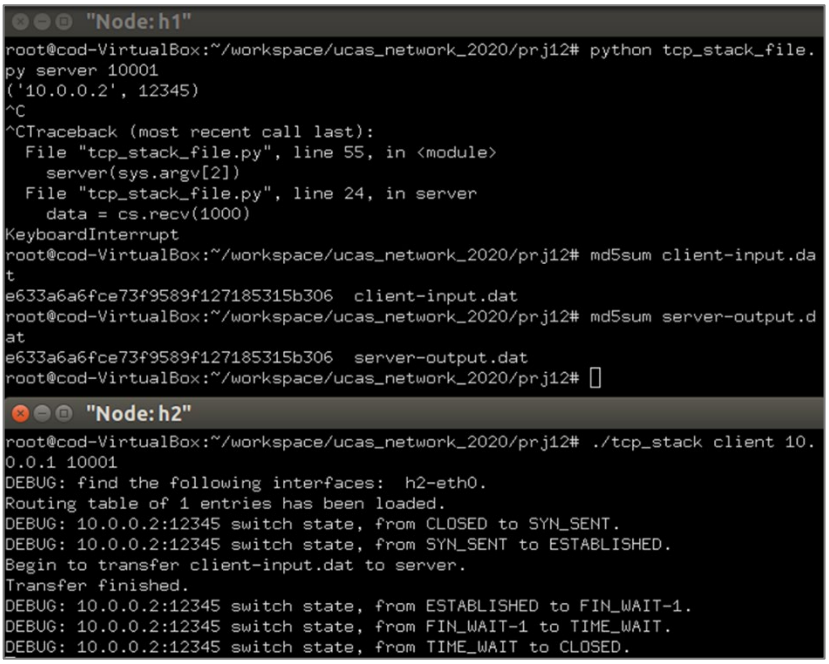


使用` md5sum `命令比较两个文件，可见结果相同，从而说明了传输内容的正确性。

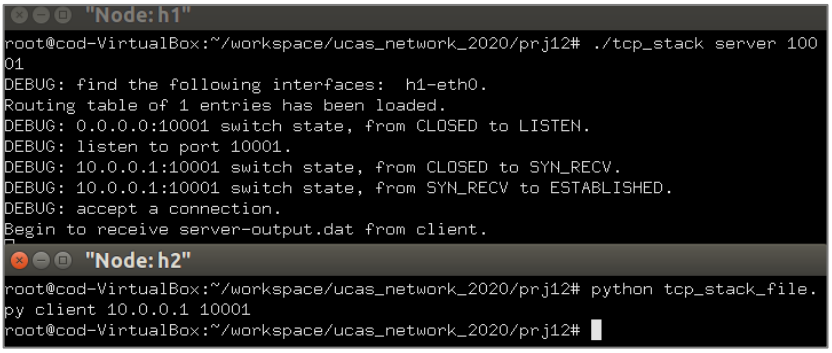


基于原脚本实现，修改得到` tcp_stack_file.py `，具体功能为用长度为 1000 字节的缓冲区接收数据并写入文件，或是以 500 字节大小的缓冲区读取文件并发送。具体代码实现较为简单，略去。**值得一提的是，500 字节的发送缓冲是考虑 MSS 限制的结果。由于 python 对于 socketAPI 的优化，超过 MSS=536 字节的数据会被分包传输。而本实验实现中未考虑分包情况，所以这里直接限制 python 脚本发送的大小来规避。**

如下图所示，用` tcp_stack_file.py `脚本替代服务器端后，连接依然正常，且 md5sum 的检查结果也正确。



如下图所示，用` tcp_stack_file.py `脚本替代客户端后，连接依然正常，且 md5sum 的检查结果也正确。



（二）分析总结

本次实验虽然只实现了基本的接收缓存相关维护，但涉及的细节同样相当丰富。让自己编写的服务器和客户端成功运行并不是件难事，但在加入对既有 API 的协同测试后，很多问题都会暴露出来。其中，让我印象最深的包括文件写回后` fflush `函数的即时刷新功能、服务器端 ACK 信号的省略以及 TCP 分包后的处理。这些都显示出之前设计中或大或小的各种漏洞和不周全，有待后续实验中继续填补并完善。