13-网络传输机制(TCP)实验二报告

陈彦帆 2018K8009918002

1、实验内容

（1）实现 TCP 数据传输：将数据封装到数据包并发送；收到数据和 ACK 时的相应处理。

（2）实现流量控制：通过调整 recv_window 来表达自己的接收能力。

（3）实现 tcp_sock 相关函数：read，write。

（4）在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送字符串，h2 echo，h1 打印收到的信息。然后，在一端用 tcp_stack.py 替换 tcp_stack 执行，测试另一端。

（5）修改 tcp_apps.c。在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送一个大小为几 MB 的文件，比较 h1 接收的文件与 h2 发送的文件是否相同。然后，在一端用修改后的 tcp_stack.py 替换 tcp_stack 执行，测试另一端。

2、实验流程

（1）完成代码编写并编译。

（2）测试正确性。

（实验内容(4)和(5)的测试流程相同，后者在测试前需要执行 bash create_randfile.sh 在测试后需要执行 diff server-output.dat client-input.dat）

首先通过 mininet 环境(tcp_topo.py)构建网络拓扑如下图：

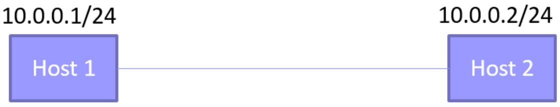


图 1

在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，进行抓包测试：

执行：

```
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# wireshark
h1# ./tcp_stack server 10001
```

```
or h1# python tcp_stack.py server 10001
h2# ./tcp_stack client 10.0.0.1 10001
or h2# python tcp_stack.py client 10.0.0.1 10001
mininet> quit
```

3、分析与实现

（1）实现 TCP 数据传输。

① 收到数据和 ACK 时的相应处理：

首先检查收到的包是否有效。收到的包的字节序必须在本端的接收窗口以内。在本实验中，不能处理乱序到达的包，这些包将被丢弃并报错。

每收到一个有效的包，更新对应的 sock 信息：更新 rcv_next 为收到数据的第一个字节加上数据长度；如果收到的包包含 ACK，更新 snd_una；根据收到包的接收窗口更新 sock 的发送窗口；如果收到的包有新的信息，发送 ACK 包。

```c
static inline int update_tsk(struct tcp_sock *tsk, struct tcp_cb *cb)
{
  tsk->rcv_nxt = cb->seq_end;
  if(cb->flags & TCP_ACK)
    tsk->snd_una = max(tsk->snd_una,cb->ack);
  tcp_update_window_safe(tsk,cb);
  if(tsk->state != TCP_LISTEN && less_than_32b(cb->seq,cb->seq_end))
    tcp_send_control_packet(tsk,TCP_ACK);
  return 0;
}
```

如果收到的包包含数据部分，则将数据部分写入环形缓存。出于简单考虑，若缓存放不下整个包，整个包将被丢弃。

环形缓存是一个生产者-消费者模型，若缓存为空，消费者被挂起。但若缓存为满，生产者不需要被挂起，而是把超出缓存的部分丢弃。

在访问缓存时，需要上锁。这里用的是 tsk->wait_recv->lock。当一方被挂起时，调用 pthread_cond_wait，将锁交出并等待唤醒。唤醒时，将尝试获取锁。

```c
void handle_tcp_data(struct tcp_sock *tsk, struct tcp_cb *cb)
{
  if(!cb->pl_len) return;
  pthread_mutex_lock(&tsk->wait_recv->lock);
  if(ring_buffer_free(tsk->rcv_buf) < cb->pl_len)
    log(DEBUG,"RECV BUFFER FULL. DROPED.");
  else{
    write_ring_buffer(tsk->rcv_buf,cb->payload,cb->pl_len);
    wake_with_lock(tsk->wait_recv);
  }
  pthread_mutex_unlock(&tsk->wait_recv->lock);
}
```

② 将数据封装到数据包并发送

框架已经实现。

（2）实现流量控制：通过调整 recv_window 来表达自己的接收能力。

每收到一个数据包，调整当前 sock 的发送窗口为对端的接收窗口大小。调整接收窗口为环形缓存的剩余大小。

（3）实现 tcp_sock 相关函数：

① read

函数调用者是消费者。若环形缓存为空，则交出锁进入等待。否则读取数据并返回。若唤醒时发现 sock 进入 CLOSE_WAIT 状态，说明发送方断开连接，返回 0。

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)
{
  pthread_mutex_lock(&tsk->wait_recv->lock);
  while(ring_buffer_empty(tsk->rcv_buf)){
    if(tsk->state == TCP_CLOSE_WAIT)
      return 0;
    sleep_with_lock(tsk->wait_recv);
  }
  int readlen = read_ring_buffer(tsk->rcv_buf,buf,len);
  tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);
  pthread_mutex_unlock(&tsk->wait_recv->lock);
  return readlen;
}
```

② write

每次发送的大小为 min(MSS,data_left,send_window)。若发送窗口为 0，则交出锁等待。否则调用 tcp_send_packet 进行发送。

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)
{
  int tot = len;
  while(len>0){
    int headerlen = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + TCP_BASE_HDR_SIZE;
    int sendlen = min(len,ETH_FRAME_LEN-headerlen);
    pthread_mutex_lock(&tsk->wait_send->lock);
    while(tsk->snd_wnd==0)
      sleep_with_lock(tsk->wait_send);
    sendlen = min(sendlen,tsk->snd_wnd);
    pthread_mutex_unlock(&tsk->wait_send->lock);
    int pkt_len = sendlen + headerlen;
    char *pkt = malloc(pkt_len);
    memcpy(pkt+headerlen,buf,sendlen);
    tcp_send_packet(tsk,pkt,pkt_len);
    buf += sendlen, len -= sendlen;
  }
  return tot;
}
```

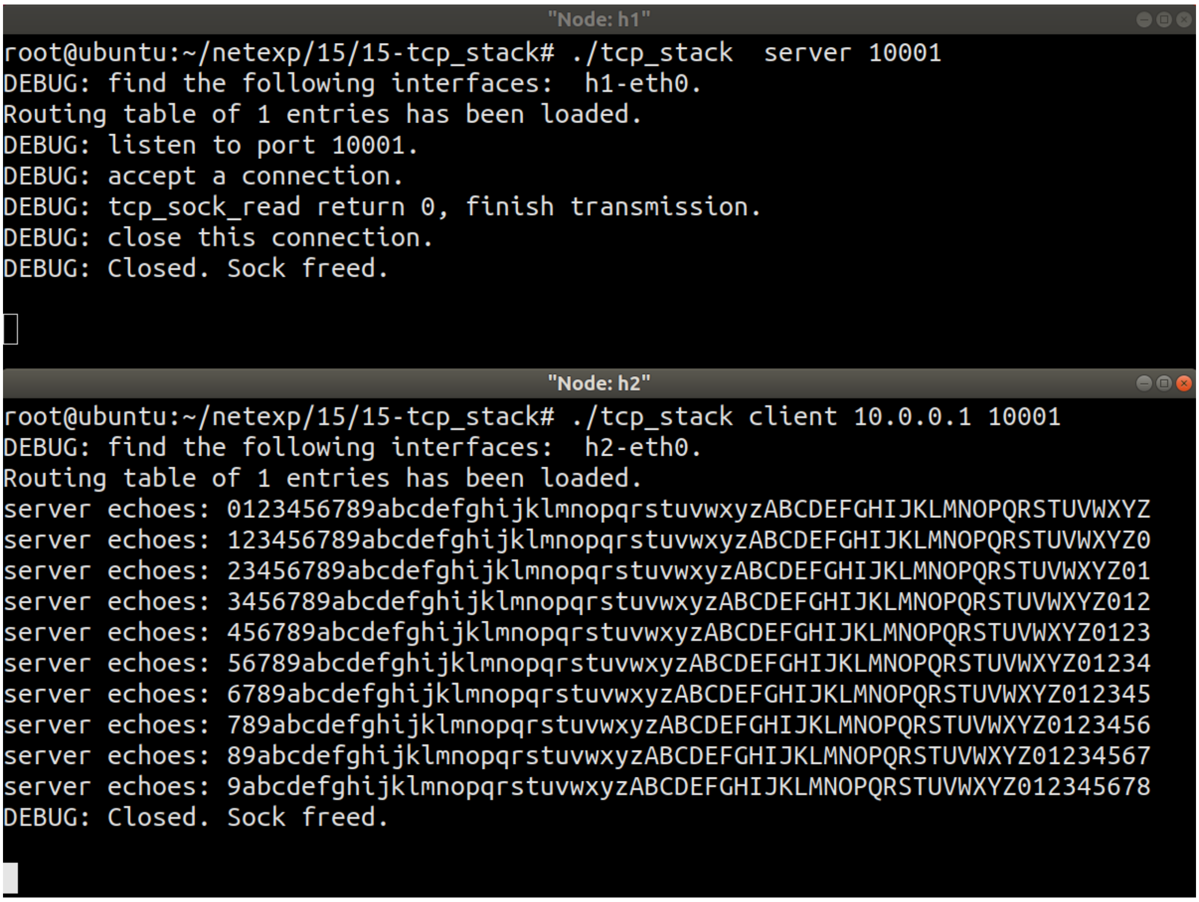
4、实验结果

网络拓扑如图 1。

（1）测试 1：字符串回显。

shell 输出结果如图 2。抓包结果如图 3。将 tcp_stack 一端替换为 python 程序后，结果如图 4-5。

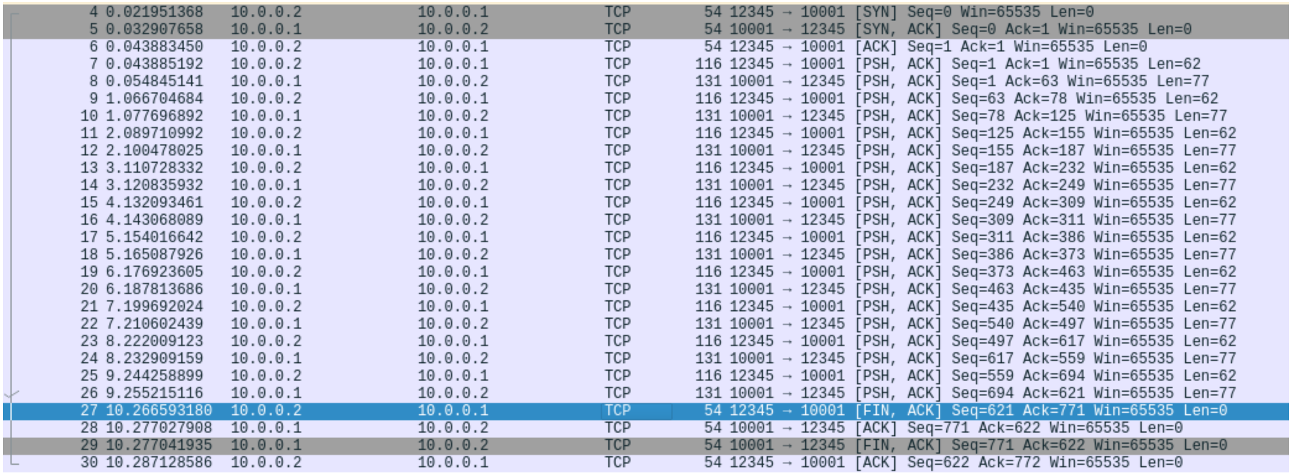
经过比对，我实现的 tcp 读写功能正确，任意一端替换为 python 程序后，结果也正确。



```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack  server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listen to port 10001.
DEBUG: accept a connection.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: Closed. Sock freed.
```

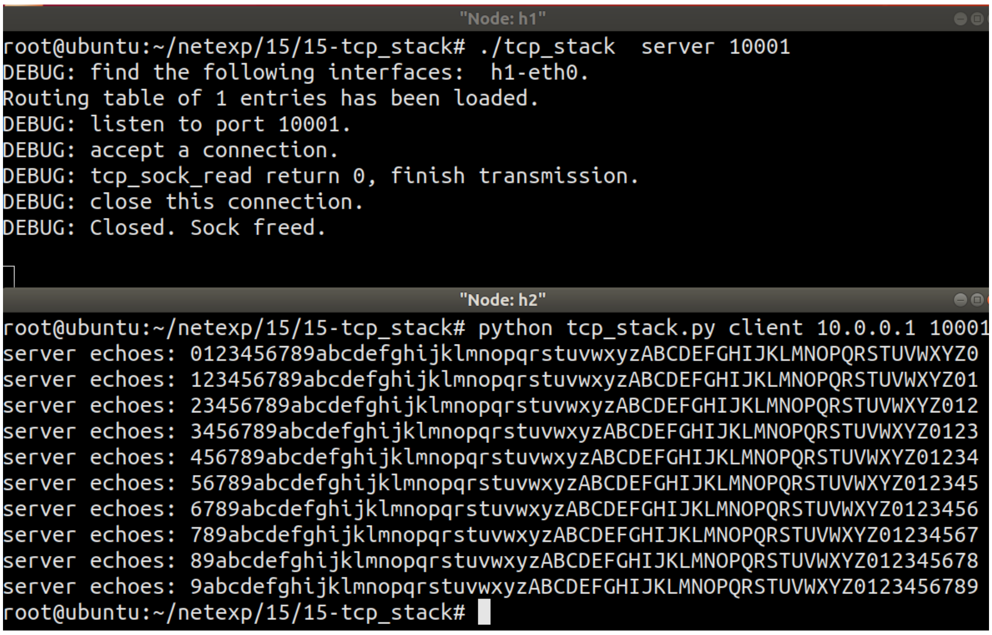
```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
DEBUG: Closed. Sock freed.
```

图 2



```
  4 0.021951368   10.0.0.2   10.0.0.1   TCP    54 12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
  5 0.032907658   10.0.0.1   10.0.0.2   TCP    54 10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
  6 0.043883450   10.0.0.2   10.0.0.1   TCP    54 12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
  7 0.043885192   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=1 Ack=1 Win=65535 Len=62
  8 0.054845141   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=1 Ack=63 Win=65535 Len=77
  9 1.066704684   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=63 Ack=78 Win=65535 Len=62
 10 1.077696892   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=78 Ack=125 Win=65535 Len=77
 11 2.089710992   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=125 Ack=155 Win=65535 Len=62
 12 2.100478025   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=155 Ack=187 Win=65535 Len=77
 13 3.110728332   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=187 Ack=232 Win=65535 Len=62
 14 3.120835932   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=232 Ack=249 Win=65535 Len=77
 15 4.132093461   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=249 Ack=309 Win=65535 Len=62
 16 4.143068089   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=309 Ack=311 Win=65535 Len=77
 17 5.154016642   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=311 Ack=386 Win=65535 Len=62
 18 5.165087926   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=386 Ack=373 Win=65535 Len=77
 19 6.176923605   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=373 Ack=463 Win=65535 Len=62
 20 6.187813686   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=463 Ack=435 Win=65535 Len=77
 21 7.199692024   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=435 Ack=540 Win=65535 Len=62
 22 7.210602439   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=540 Ack=497 Win=65535 Len=77
 23 8.222009123   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=497 Ack=617 Win=65535 Len=62
 24 8.232909159   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=617 Ack=559 Win=65535 Len=77
 25 9.244258899   10.0.0.2   10.0.0.1   TCP   116 12345 → 10001 [PSH, ACK] Seq=559 Ack=694 Win=65535 Len=62
 26 9.255215116   10.0.0.1   10.0.0.2   TCP   131 10001 → 12345 [PSH, ACK] Seq=694 Ack=621 Win=65535 Len=77
 27 10.266593180  10.0.0.2   10.0.0.1   TCP    54 12345 → 10001 [FIN, ACK] Seq=621 Ack=771 Win=65535 Len=0
 28 10.277027908  10.0.0.1   10.0.0.2   TCP    54 10001 → 12345 [ACK] Seq=771 Ack=622 Win=65535 Len=0
 29 10.277041935  10.0.0.1   10.0.0.2   TCP    54 10001 → 12345 [FIN, ACK] Seq=771 Ack=622 Win=65535 Len=0
 30 10.287128586  10.0.0.2   10.0.0.1   TCP    54 12345 → 10001 [ACK] Seq=622 Ack=772 Win=65535 Len=0
```

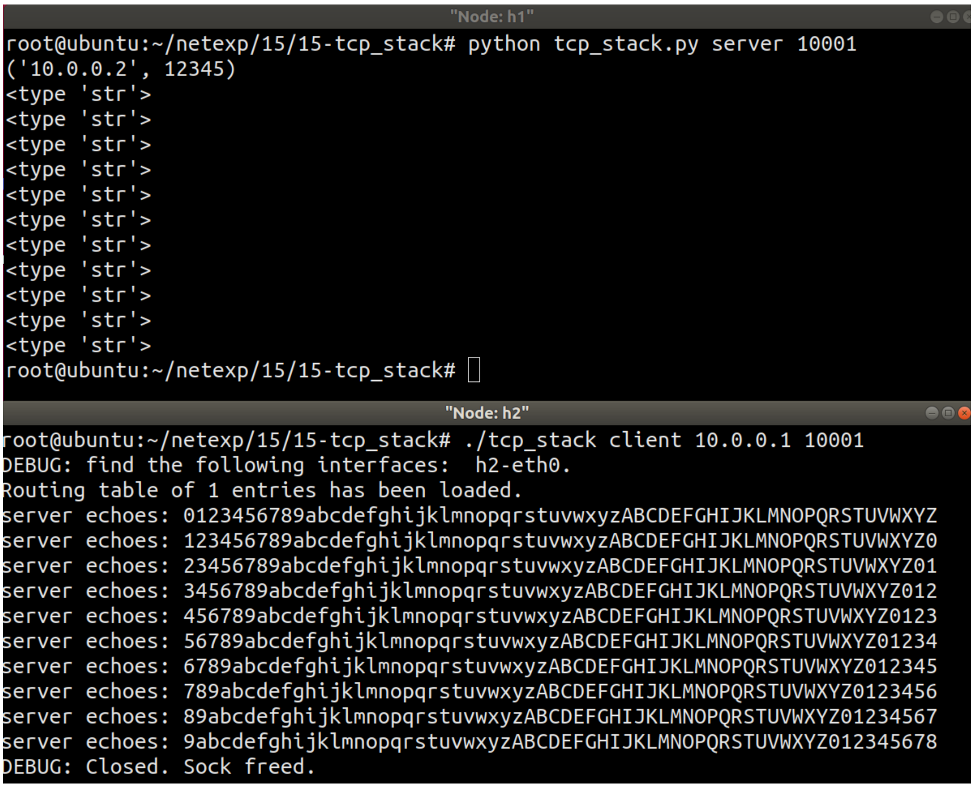
图 3



```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack  server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listen to port 10001.
DEBUG: accept a connection.
DEBUG: tcp_sock_read return 0, finish transmission.
DEBUG: close this connection.
DEBUG: Closed. Sock freed.
```

```
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack.py client 10.0.0.1 10001
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
root@ubuntu:~/netexp/15/15-tcp_stack#
```

图 4 client python



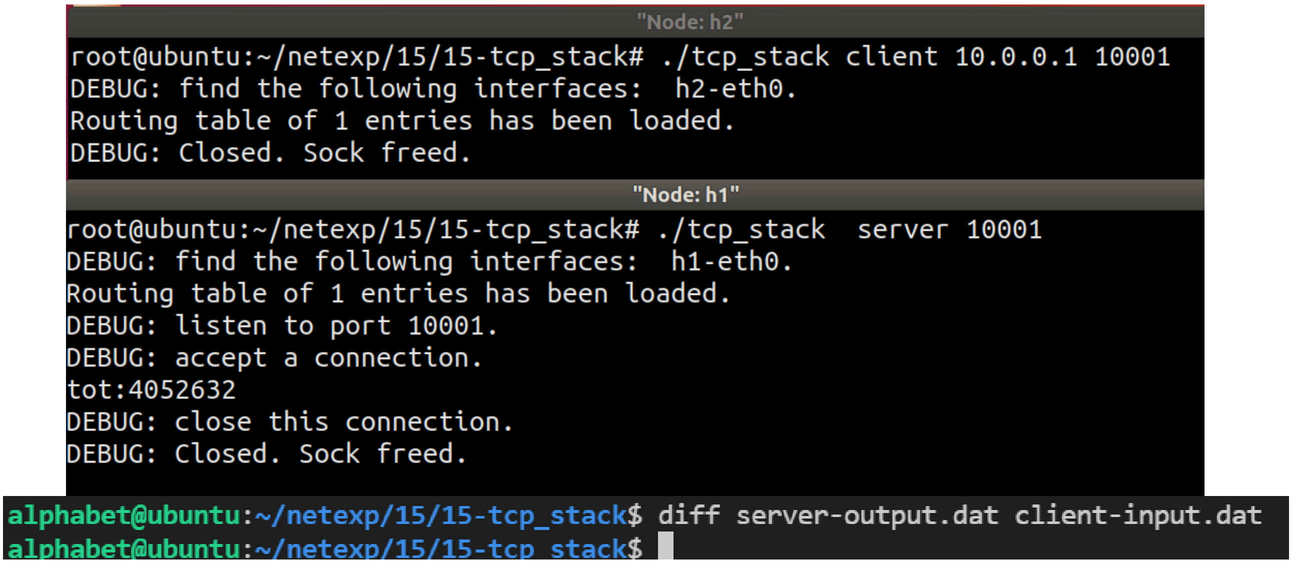
```
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack.py server 10001
('10.0.0.2', 12345)
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
root@ubuntu:~/netexp/15/15-tcp_stack# 
```

```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
DEBUG: Closed. Sock freed.
```

图 5 server python

（2）测试 2：上传大文件（约 4MB）

shell 输出结果如图 6。传输后，通过 diff 命令比对，接收方收到的文件与发送方上传的完全相同。将 tcp_stack 一端替换为 python 程序后，结果如图 7-8。同样通过了 diff 比对。

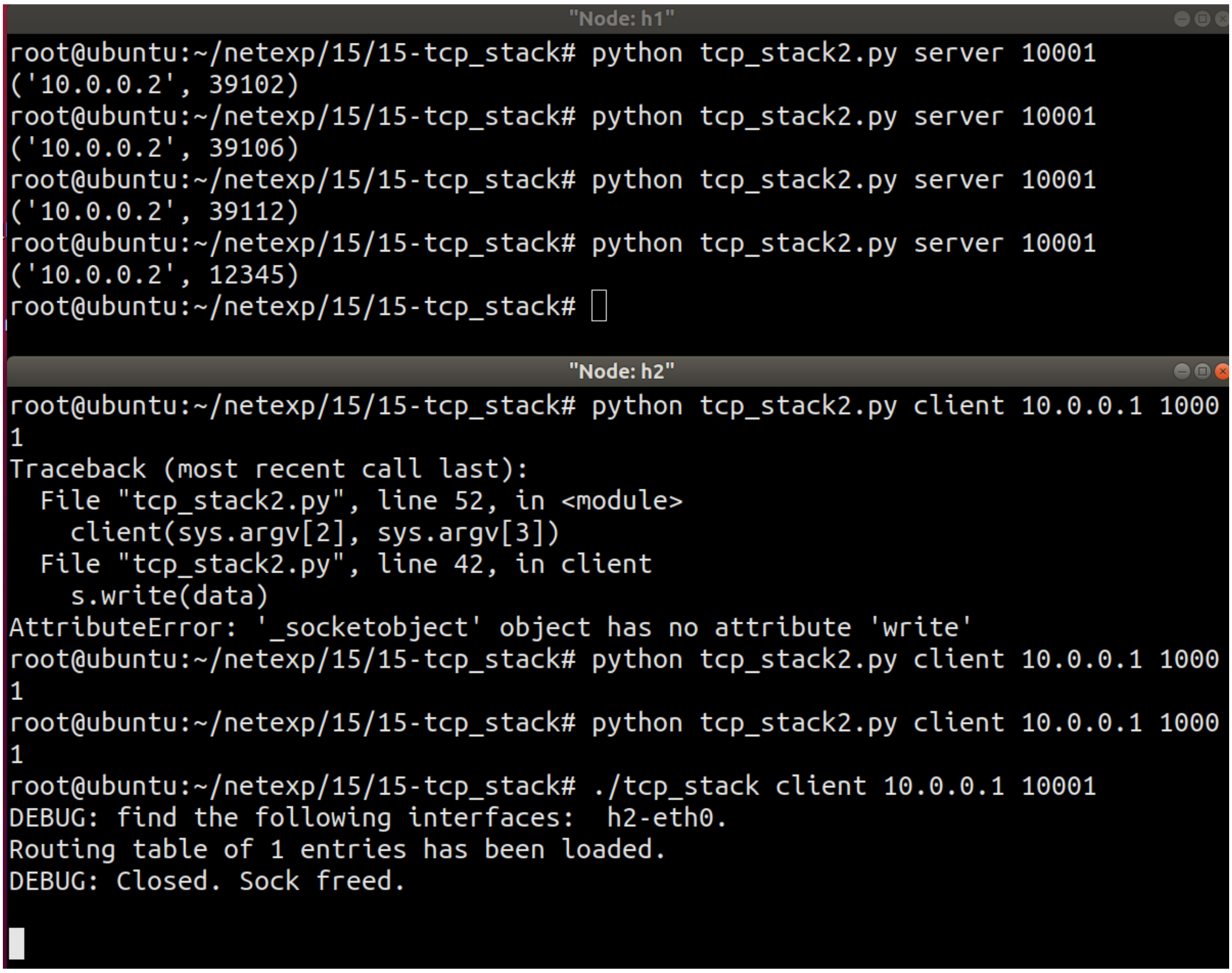


```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: Closed. Sock freed.
```

```
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack  server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listen to port 10001.
DEBUG: accept a connection.
tot:4052632
DEBUG: close this connection.
DEBUG: Closed. Sock freed.
```

```
alphabet@ubuntu:~/netexp/15/15-tcp_stack$ diff server-output.dat client-input.dat
alphabet@ubuntu:~/netexp/15/15-tcp_stack$ 
```

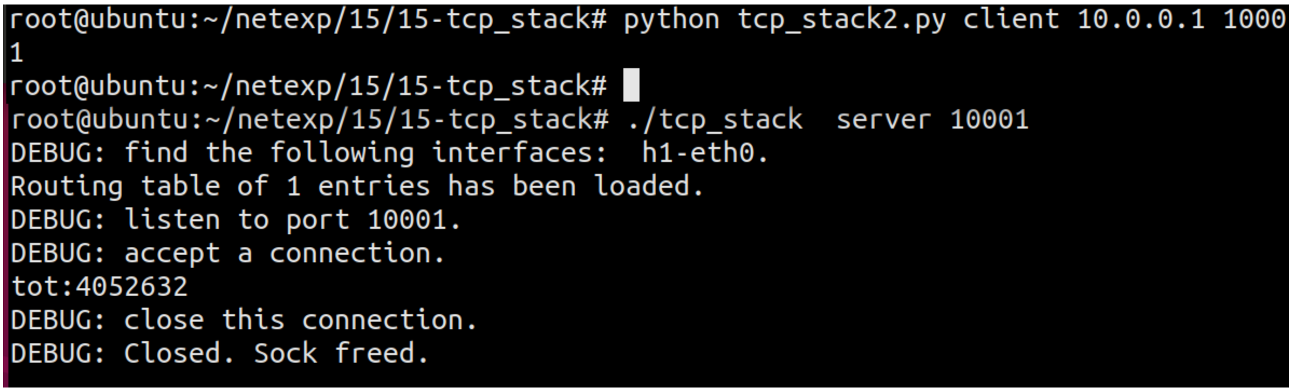
图 6



```
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py server 10001
('10.0.0.2', 39102)
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py server 10001
('10.0.0.2', 39106)
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py server 10001
('10.0.0.2', 39112)
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py server 10001
('10.0.0.2', 12345)
root@ubuntu:~/netexp/15/15-tcp_stack# 
```

```
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py client 10.0.0.1 1000
1
Traceback (most recent call last):
  File "tcp_stack2.py", line 52, in <module>
    client(sys.argv[2], sys.argv[3])
  File "tcp_stack2.py", line 42, in client
    s.write(data)
AttributeError: '_socketobject' object has no attribute 'write'
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py client 10.0.0.1 1000
1
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py client 10.0.0.1 1000
1
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack client 10.0.0.1 10001
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
DEBUG: Closed. Sock freed.
```

图 7 python server



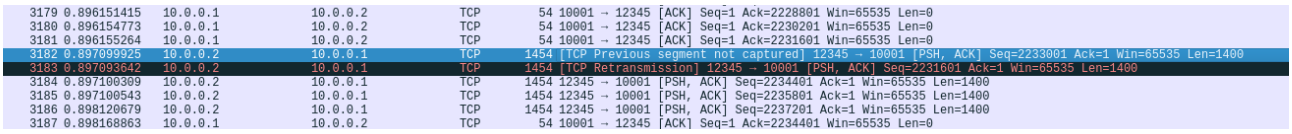
```
root@ubuntu:~/netexp/15/15-tcp_stack# python tcp_stack2.py client 10.0.0.1 1000
1
root@ubuntu:~/netexp/15/15-tcp_stack# 
root@ubuntu:~/netexp/15/15-tcp_stack# ./tcp_stack  server 10001
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: listen to port 10001.
DEBUG: accept a connection.
tot:4052632
DEBUG: close this connection.
DEBUG: Closed. Sock freed.
```

图 8 python client

5、遇到的问题

（1）out of order

若发送方发送间隔小于 RT prop，可能会造成数据包乱序，相应的抓包结果如图 9。本实验无法处理乱序包，故将发送间隔调大，并调小模拟器的 RTT。



```
3179 0.896151415  10.0.0.1    10.0.0.2    TCP    54 10001 → 12345 [ACK] Seq=1 Ack=2228801 Win=65535 Len=0
3180 0.896154773  10.0.0.1    10.0.0.2    TCP    54 10001 → 12345 [ACK] Seq=1 Ack=2230201 Win=65535 Len=0
3181 0.896155264  10.0.0.1    10.0.0.2    TCP    54 10001 → 12345 [ACK] Seq=1 Ack=2231601 Win=65535 Len=0
3182 0.897099925  10.0.0.2    10.0.0.1    TCP  1454 [TCP Previous segment not captured] 12345 → 10001 [PSH, ACK] Seq=2233001 Ack=1 Win=65535 Len=1400
3183 0.897093642  10.0.0.2    10.0.0.1    TCP  1454 [TCP Retransmission] 12345 → 10001 [PSH, ACK] Seq=2231601 Ack=1 Win=65535 Len=1400
3184 0.897100309  10.0.0.2    10.0.0.1    TCP  1454 12345 → 10001 [PSH, ACK] Seq=2234401 Ack=1 Win=65535 Len=1400
3185 0.897100543  10.0.0.2    10.0.0.1    TCP  1454 12345 → 10001 [PSH, ACK] Seq=2235801 Ack=1 Win=65535 Len=1400
3186 0.898120679  10.0.0.2    10.0.0.1    TCP  1454 12345 → 10001 [PSH, ACK] Seq=2237201 Ack=1 Win=65535 Len=1400
3187 0.898168863  10.0.0.1    10.0.0.2    TCP    54 10001 → 12345 [ACK] Seq=1 Ack=2234401 Win=65535 Len=0
```

图 9

（2）socket raw send buffer FULL

若发送过快，socket 可能会出现如下提示：

Send raw packet failed: No buffer space available

解决方法：增加发送间隔。

（3）recv buffer FULL

若发送过快，发送方尚未收到接收方的 ACK 包时已经发送了大量数据，来不及调整发送窗口，造成接收方缓存爆满。解决方法：解决方法：增加发送间隔。

（4）文件写入失败

接收方在向本地写文件完毕后，应调用 fclose，否则文件可能写入失败。

（5）ACK 超时

接收方的 ACK 可与数据一并发送。在本实验中，client 间隔 1s 发送一次字符串，此时 ACK 已经超时，python server 会重传。解决办法：收到数据包后马上发送 ACK。

（6）唤醒条件不满足：

对于框架给出的 sleep_on 和 wake_up 函数，若先调用了 wake_up，则调用 sleep_on 时不会睡眠而是立即返回，不满足唤醒条件，造成错误。

解决方法：重写 sleep_on 和 wake_up 函数，使 sleep_on 只有在调用后被唤醒才会返回。