13-网络传输机制(TCP)实验一报告

陈彦帆 2018K8009918002

1、实验内容

（1）基于已有代码框架，实现 TCP Sock 的连接管理相关函数。

（2）在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送连接请求和关闭连接请求。通过 wireshark 抓包测试正确性。然后，在一端用 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。

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）TCP 连接管理

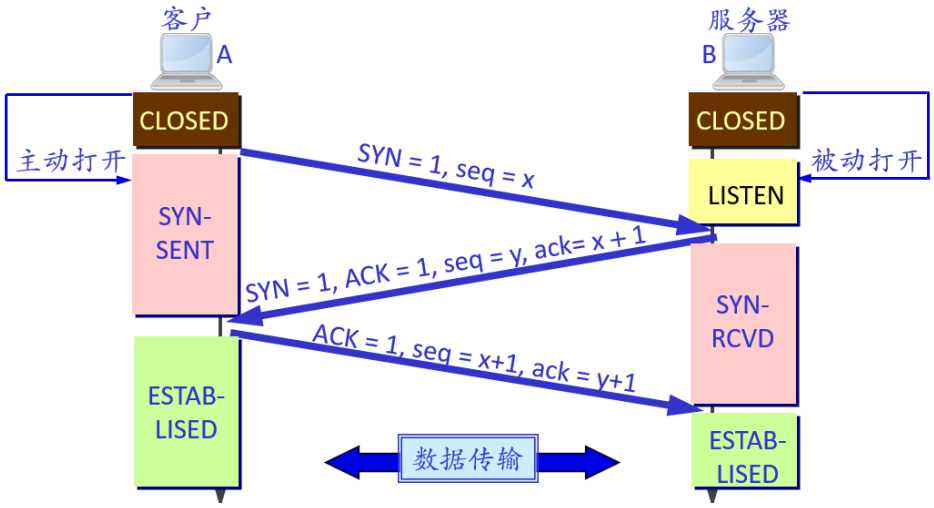


图 2 TCP 连接示意图

服务端连接流程：

（1）bind：绑定源地址和源端口，哈希到 bind_table。

（2）listen：把源地址和源端口，哈希到 listen_table，监听具有相同目标地址和端口的 TCP 请求。

（3）listen 阶段收到 SYN：进入 SYN_RCVD 阶段，创建一个具有完整四元组的子 sock，哈希到 established_table，并加入父 sock 的 listen_queue。（父 sock 继续监听）

（4）SYN_RCVD 阶段收到第三次握手：把当前 sock(子 sock)从父 sock 的 listen_queue 转移至 established_queue，建立连接。转移的时候，可能与正在 accept 的应用程序冲突，需要上锁。转移后，唤醒 accept 线程（如果有）。进入 ESTABLISHED 阶段。

每收到一个更新的 TCP 包，需要更新 tsk->rcv_nxt = cb->seq_end; tsk->snd_una = cb->ack;

对应的代码：

```
else if(tsk->state == TCP_LISTEN && thr->flags & TCP_SYN){
  struct tcp_sock *csk = alloc_tcp_sock();
  csk->sk_sip = cb->daddr;
  csk->sk_sport = cb->dport;
  csk->sk_dip = cb->saddr;
  csk->sk_dport = cb->sport;
  csk->rcv_nxt = cb->seq_end;
  csk->parent = tsk;
  csk->state = TCP_SYN_RECV;
  pthread_mutex_lock(&tsk->wait_accept->lock);
  tcp_sock_listen_enqueue(csk);
  pthread_mutex_unlock(&tsk->wait_accept->lock);
  tcp_hash(csk);
```

```
    tcp_send_control_packet(csk,TCP_SYN|TCP_ACK);
  }
  else if(tsk->state == TCP_SYN_RECV && thr->flags & TCP_ACK){
    struct tcp_sock *psk = tsk->parent;
    pthread_mutex_lock(&psk->wait_accept->lock);
    if(!tcp_sock_accept_queue_full(psk)){
      tsk->state = TCP_ESTABLISHED;
      update_tsk(tsk,cb);
      tcp_sock_accept_enqueue(tsk);
      wake_with_lock(psk->wait_accept);
    }
    pthread_mutex_unlock(&psk->wait_accept->lock);
  }
...
int tcp_sock_bind(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
  // omit the ip address, and only bind the port
  tsk->sk_sip = 0;
  int ret = tcp_sock_set_sport(tsk, ntohs(skaddr->port));
  tcp_bind_hash(tsk);
  return ret;
}
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
  tsk->backlog = backlog;
  tsk->state = TCP_LISTEN;
  return tcp_hash(tsk);
}
```

客户端连接流程：

（1）connect：设置源 IP 地址和源端口，哈希到 bind_table。设置四元组，哈希到 established_table。发送 SYN 请求，进入 SYN_SENT 阶段，sleep 等待应答。

（2）在 SYN_SENT 收到 ACK|SYN：发送 ack 包，进入 ESTABLISHED 阶段，唤醒 connect 线程。

对应的代码：

```
  else if(tsk->state == TCP_SYN_SENT && thr->flags & (TCP_ACK|TCP_SYN)){
    update_tsk(tsk,cb);
    tsk->state = TCP_ESTABLISHED;
    tcp_send_control_packet(tsk,TCP_ACK);
    wake_up(tsk->wait_connect);
  }
```

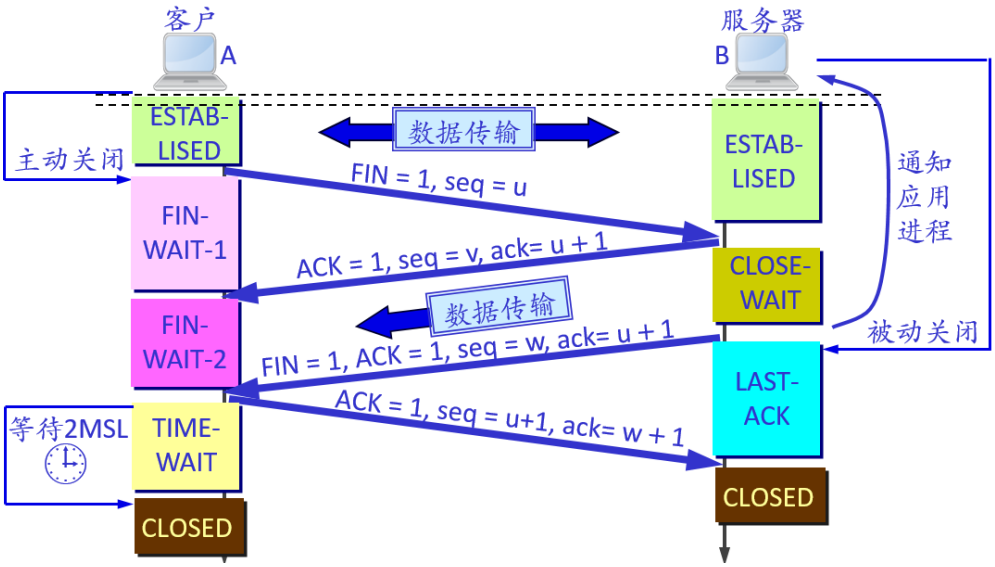


图 3 TCP 断开连接示意图

服务端断开连接流程：

（1）在 ESTABLISHED 阶段收到 FIN 包：进入 CLOSE_WAIT 阶段，发送 ACK 包。通知应用进程（在本实验中，等待应用线程调用 close）。

（2）在 CLOSE_WAIT 阶段调用 close：发送 FIN|ACK 第三次挥手。进入 LAST_ACK 阶段。

（3）在 LAST_ACK 阶段收到第四次挥手：状态转为 CLOSED，释放子 sock 的资源（unhash_established_table，unhash_bind_table）。

客户端断开连接流程：

（1）调用 close：进入 TCP_FIN_WAIT_1，发送 FIN 包。

（2）在 TCP_FIN_WAIT_1 收到 ACK 包：进入 TCP_FIN_WAIT_2。

（3）在 TCP_FIN_WAIT_2 或 TCP_FIN_WAIT_1 收到 FIN|ACK 包：进入 TIME_WAIT，发送第四次握手，将该 sock 加入 timewait 队列，等待 2MSL 后进入 CLOSED 并释放资源（unhash_established_table，unhash_bind_table）。

对应的代码：

```
void tcp_sock_close(struct tcp_sock *tsk)
{
  if(tsk->state == TCP_ESTABLISHED){
    tsk->state = TCP_FIN_WAIT_1;
    tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
  }
  else if(tsk->state == TCP_CLOSE_WAIT){
    tsk->state = TCP_LAST_ACK;
    tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
  }
  else{
```

```
    tcp_send_control_packet(tsk, TCP_RST);
    tsk->state = TCP_CLOSED;
    tcp_unhash(tsk);
    tcp_bind_unhash(tsk);
  }
}
...
  else if(tsk->state == TCP_ESTABLISHED && thr->flags & TCP_FIN){
    tsk->state = TCP_CLOSE_WAIT;
    update_tsk(tsk,cb);
    tcp_send_control_packet(tsk,TCP_ACK);
  }
  else if(tsk->state == TCP_FIN_WAIT_1 && thr->flags & TCP_ACK){
    tsk->state = TCP_FIN_WAIT_2;
    if(thr->flags & TCP_FIN)
      tsk->state = TCP_TIME_WAIT;
    update_tsk(tsk,cb);
  }
  else if(tsk->state == TCP_FIN_WAIT_2 && thr->flags & (TCP_FIN|TCP_ACK)){
    tsk->state = TCP_TIME_WAIT;
    update_tsk(tsk,cb);
    tcp_send_control_packet(tsk,TCP_ACK);
    tcp_set_timewait_timer(tsk);
  }
  else if(tsk->state == TCP_LAST_ACK && thr->flags & (TCP_ACK)){
    tsk->state = TCP_CLOSED;
    update_tsk(tsk,cb);
    tcp_unhash(tsk);
  }
```

4、实验结果

网络拓扑如图 1。shell 输出结果如图 4。在一端用 tcp_stack.py 替换 tcp_stack 执行后，抓包结果如图 5-7。（测试时，server 在 CLOSE_WAIT 阶段直接发送 close，没有等待 5s）

经过比对，我实现的 TCP 连接和断开功能正确。

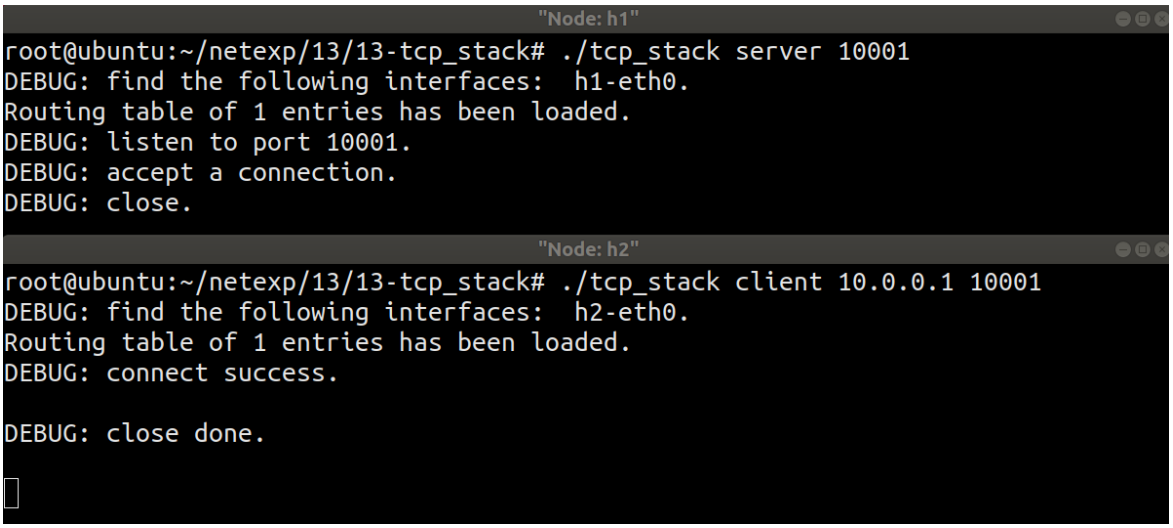
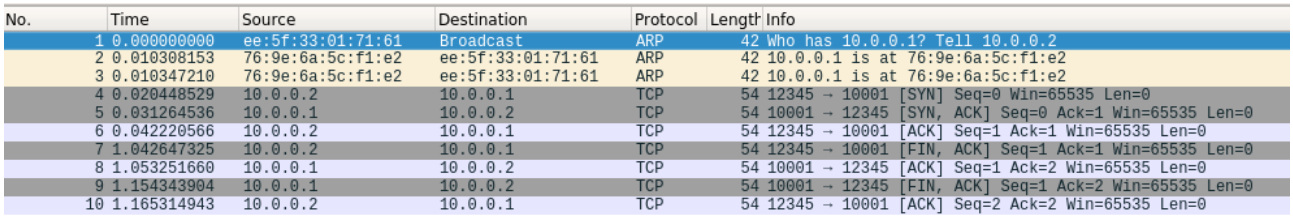
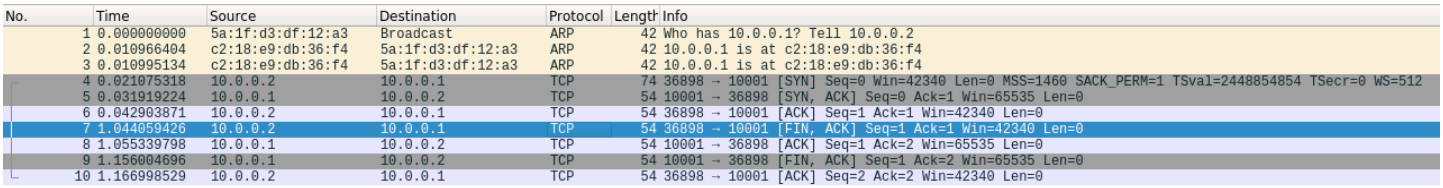


图 4 shell 输出结果



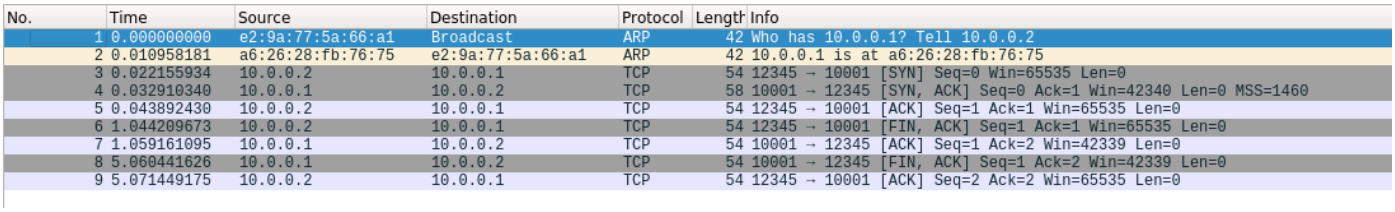
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	ee:5f:33:01:71:61	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010308153	76:9e:6a:5c:f1:e2	ee:5f:33:01:71:61	ARP	42	10.0.0.1 is at 76:9e:6a:5c:f1:e2
3	0.010347210	76:9e:6a:5c:f1:e2	ee:5f:33:01:71:61	ARP	42	10.0.0.1 is at 76:9e:6a:5c:f1:e2
4	0.020448529	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.031264536	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.042220566	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.042647325	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.053251660	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	1.154343904	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	1.165314943	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

图 5 both mine



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	5a:1f:d3:df:12:a3	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010966404	c2:18:e9:db:36:f4	5a:1f:d3:df:12:a3	ARP	42	10.0.0.1 is at c2:18:e9:db:36:f4
3	0.010995134	c2:18:e9:db:36:f4	5a:1f:d3:df:12:a3	ARP	42	10.0.0.1 is at c2:18:e9:db:36:f4
4	0.021075318	10.0.0.2	10.0.0.1	TCP	74	36898 → 10001 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=2448854854 TSecr=0 WS=512
5	0.031919224	10.0.0.1	10.0.0.2	TCP	54	10001 → 36898 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.042903871	10.0.0.2	10.0.0.1	TCP	54	36898 → 10001 [ACK] Seq=1 Ack=1 Win=42340 Len=0
7	1.044059426	10.0.0.2	10.0.0.1	TCP	54	36898 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=42340 Len=0
8	1.055339798	10.0.0.1	10.0.0.2	TCP	54	10001 → 36898 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	1.156004696	10.0.0.1	10.0.0.2	TCP	54	10001 → 36898 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	1.166998529	10.0.0.2	10.0.0.1	TCP	54	36898 → 10001 [ACK] Seq=2 Ack=2 Win=42340 Len=0

图 6 server mine, client python



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	e2:9a:77:5a:66:a1	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010958181	a6:26:28:fb:76:75	e2:9a:77:5a:66:a1	ARP	42	10.0.0.1 is at a6:26:28:fb:76:75
3	0.022155934	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
4	0.032910340	10.0.0.1	10.0.0.2	TCP	58	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=42340 Len=0 MSS=1460
5	0.043892430	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
6	1.044209673	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.059161095	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=42339 Len=0
8	5.060441626	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=42339 Len=0
9	5.071449175	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

图 7 client mine, server python

5、遇到的问题

（1）给定的框架中 tsk->listen_queue， tsk->established_queue，bind_table，listen_table， established_table， timewait_table 都可能出现多线程同时访问，需要上锁。其中，accept 函数实现如下：

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
  pthread_mutex_lock(&tsk->wait_accept->lock);
  while(list_empty(&tsk->accept_queue))
    sleep_with_lock(tsk->wait_accept);
  struct tcp_sock *csk = tcp_sock_accept_dequeue(tsk);
  pthread_mutex_unlock(&tsk->wait_accept->lock);
  return csk;
}
```

accept_queue 是一个生产者-消费者模型，生产者是 tcp 协议栈，消费者是应用程序。消费者判空，出队操作需要带锁，同样地，生产者判满，入队操作也需要上锁。若条件不满足，调用 cond_wait 释放锁，等待 signal 后重新获取锁。

```
  else if(tsk->state == TCP_SYN_RECV && thr->flags & TCP_ACK){
    struct tcp_sock *psk = tsk->parent;
    pthread_mutex_lock(&psk->wait_accept->lock);
    if(!tcp_sock_accept_queue_full(psk)){
      tsk->state = TCP_ESTABLISHED;
      update_tsk(tsk,cb);
      tcp_sock_accept_enqueue(tsk);
      wake_with_lock(psk->wait_accept);
    }
    pthread_mutex_unlock(&psk->wait_accept->lock);
  }
```

其中，sleep_with_lock 函数为框架中的 sleep_on 函数去除头尾的获取、释放锁操作后得到的函数。wait_with_lock 同理。

在本实验中，需要加锁的数据结构较多，为了避免死锁，在编写代码时，每个线程只允许最多同时占有一个锁。

（2）当 sock 关闭后，会释放资源。此时若应用程序再次调用 close，会引起段错误。可能需要以 sock 编号代替 sock 指针作为应用程序调用的接口。

（3）IP 实现采用框架给出的模块。测试时，会出现 ARP 重复响应的情况（如图 5-6），暂不清楚原因。

（4）tcp_sock_lookup_listen 时，sip 应设置为 0，才能找到 sip 被设为 0 的正在监听的 sock。