16-网络传输机制(TCP)实验三报告

陈彦帆 2018K8009918002

1、实验内容

（1）在上个实验的基础上，实现基于超时重传的 TCP 可靠数据传输，使得节点之间在有丢包网络中能够建立连接并正确传输数据。

（2）在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，在丢包率为 2%的链路上向 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。

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：

```
bash create_randfile.sh
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# wireshark
h1# ./tcp_stack server 10001
or h1# python tcp_stack2.py server 10001
h2# ./tcp_stack client 10.0.0.1 10001
or h2# python tcp_stack2.py client 10.0.0.1 10001
mininet> quit
diff server-output.dat client-input.dat
```

3、分析与实现

（1）超时重传实现。

① 发送一个新的数据包时(携带数据或 SYN|FIN)：

将该数据包放进发送队列末尾；

如果定时器未启动，则启动定时器。

```
void tcp_send_packet(struct tcp_sock *tsk, char *packet, int len)
...
  tcp_send_buf_append(tsk,packet,len);
  tcp_set_retrans_timer(tsk);
...
void tcp_send_control_packet(struct tcp_sock *tsk, u8 flags)
...
  if (flags & (TCP_SYN|TCP_FIN)){
    tsk->snd_nxt += 1;
    tcp_send_buf_append(tsk,packet,pkt_size);
    tcp_set_retrans_timer(tsk);
  }
...
```

每个 sock 有一个链表实现的发送队列，定时器和收发包线程同时访问，需要用锁维护。发送队列的每一项包括 packet，len，retrans_times。

```
void tcp_send_buf_append(struct tcp_sock *tsk, char* packet, int len)
{
  tcp_send_buf_entry_t* entry = malloc(sizeof(tcp_send_buf_entry_t));
  entry->packet = malloc(len);
  entry->len = len;
  entry->retrans_times = 0;
  memcpy(entry->packet,packet,len);
  pthread_mutex_lock(&tsk->send_buf_lock);
  list_add_tail(&entry->list,&tsk->send_buf);
  pthread_mutex_unlock(&tsk->send_buf_lock);
}
```

启动定时器时，通过检查定时器的 enable 域判断是否已经启动，如果已经启动，则无须其它操作。

定时器链表可能有收发线程进行 unset 操作，定时器线程进行扫描操作，访问时需要加锁。

```
void tcp_set_retrans_timer(struct tcp_sock *tsk)
{
  if(!tsk->retrans_timer.enable){
    tsk->retrans_timer.enable = 1;
```

```
    tsk->retrans_timer.type = 0;
    pthread_mutex_lock(&retrans_timer_lock);
    list_add_tail(&tsk->retrans_timer.list, &retrans_timer_list);
    pthread_mutex_unlock(&retrans_timer_lock);
    tsk->retrans_timer.timeout = TCP_RETRANS_TIME;
  }
}
```

② 定时器超时时

扫描当前 sock 的发送队列，发出第一个数据包。如果该数据包的重传次数超过 3，返回失败并发送 RST。否则进行重传，重传时，需要更新 ACK 和重传次数，定时器时间翻倍。

```
int tcp_send_buf_retrans(struct tcp_sock *tsk)
{
  pthread_mutex_lock(&tsk->send_buf_lock);
  tcp_send_buf_entry_t *entry;
  int times = 1;
  if(list_empty(&tsk->send_buf)) return 1;
  entry = list_entry(tsk->send_buf.next,tcp_send_buf_entry_t,list);
  if(entry->retrans_times>3){
    pthread_mutex_unlock(&tsk->send_buf_lock);
    return 0;
  }
  int len = entry->len;
  times = ++entry->retrans_times;
  char *packet = malloc(len);
  memcpy(packet,entry->packet,len);
  struct tcphdr *tcp = packet_to_tcp_hdr(packet);
  struct iphdr *ip = packet_to_ip_hdr(packet);
  tcp->ack = htonl(tsk->rcv_nxt);
  tcp->checksum = tcp_checksum(ip, tcp);
  ip_send_packet(packet,len);
  pthread_mutex_unlock(&tsk->send_buf_lock);
  return times;
}
```

③ 收到新的 ACK 时

重置当前定时器时间。（如果定时器未开启，则无须操作）

扫描发送队列，移除 seq_end 小于当前 ACK 的项。若发送队列清空，则关闭定时器。

```
int tcp_send_buf_clear(struct tcp_sock *tsk, u32 ack)
{
  pthread_mutex_lock(&tsk->send_buf_lock);
  tcp_send_buf_entry_t *entry,*q;
  list_for_each_entry_safe(entry,q,&tsk->send_buf,list){
    struct tcphdr *tcp = packet_to_tcp_hdr(entry->packet);
```

```
    struct iphdr *ip = packet_to_ip_hdr(entry->packet);
    u32 len = ntohs(ip->tot_len) - IP_HDR_SIZE(ip) - TCP_HDR_SIZE(tcp);
    u32 seq_end = ntohl(tcp->seq) + len + ((tcp->flags & (TCP_SYN|TCP_FIN)) ? 1 : 0
);
    if(less_or_equal_32b(seq_end,ack)){
      list_delete_entry(&entry->list);
      free(entry->packet);
      free(entry);
    }
    else break;
  }
  int ret = list_empty(&tsk->send_buf);
  pthread_mutex_unlock(&tsk->send_buf_lock);
  return ret;
}
```

关闭定时器时需要获取定时器链表的锁，注意锁的获取要在扫描发送队列之前，以免与扫描定时器线程发生死锁。

```
static void handle_ack(struct tcp_sock *tsk, struct tcp_cb *cb)
{
  if(cb->flags & TCP_ACK)
    if(less_than_32b(tsk->snd_una,cb->ack)){
      pthread_mutex_lock(&retrans_timer_lock);
      tsk->snd_una = cb->ack;
      tcp_reset_retrans_timer(tsk);
      if(tcp_send_buf_clear(tsk,tsk->snd_una)){ /
        tcp_unset_retrans_timer(tsk);
      }
      pthread_mutex_unlock(&retrans_timer_lock);
    }
  tcp_update_window(tsk,cb);
  if(tsk->state != TCP_LISTEN && less_than_32b(cb->seq,cb->seq_end))
    tcp_send_control_packet(tsk,TCP_ACK);
}
```

④ 收到旧的包时

可能是不必要的重传，也可能是 ACK 丢失。如果包含数据，则回复 ACK。

```
  if(!less_or_equal_32b(tsk->rcv_nxt, cb->seq_end)||
      (tsk->rcv_nxt==cb->seq_end && less_than_32b(cb->seq,cb->seq_end))){
    log(ERROR, "Dup pkt.DROP.");
    tcp_send_control_packet(tsk,TCP_ACK);
    return 0;
  }
```

（2）处理乱序的数据包

如果收到的数据包是有效的（位于接收窗口内），但为乱序，则放入 ofo_buf。否则，将数据中新的部分写入 rcv_buf，然后检查是否可以读取 ofo_buf。如果乱序的数据包含 FIN，处理较为繁琐，直接丢弃。

```
static void handle_tcp_data(struct tcp_sock *tsk, struct tcp_cb *cb)
{
  int len = (int)(cb->seq_end-tsk->rcv_nxt) - (cb->flags&(TCP_SYN|TCP_FIN)?1:0);
  if(!cb->pl_len || len<=0) {update_tsk(tsk,cb); 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{
    if(less_or_equal_32b(cb->seq, tsk->rcv_nxt)){
      write_ring_buffer(tsk->rcv_buf,cb->payload+cb->pl_len-len,len);
      update_tsk(tsk,cb);
      tcp_check_ofo_buf(tsk);
      wake_with_lock(tsk->wait_recv);
      tsk->rcv_wnd = max(ring_buffer_free(tsk->rcv_buf),1);
    }
    else
      tcp_ofo_buf_append(tsk,cb);
  }
  pthread_mutex_unlock(&tsk->wait_recv->lock);
}
```

写入 ofo_buf 时，按 seq_end 组成优先队列。

```
void tcp_ofo_buf_append(struct tcp_sock *tsk, struct tcp_cb *cb)
{
  assert((cb->flags&TCP_FIN)==0);
  tcp_ofo_buf_entry_t *entry,*q, *new_entry;
  list_for_each_entry_safe(entry,q,&tsk->rcv_ofo_buf,list){
    if(less_or_equal_32b(cb->seq_end,entry->seq + entry->len)){
      if(greater_or_equal_32b(cb->seq,entry->seq))
        return;
      break;
    }
  }
  new_entry = malloc(sizeof(tcp_ofo_buf_entry_t));
  new_entry->seq = cb->seq;
  new_entry->len = cb->pl_len;
  new_entry->data = malloc(cb->pl_len);
  memcpy(new_entry->data,cb->payload,new_entry->len);
  list_add_tail(&new_entry->list,entry->list.prev);
}
```

读取 ofo_buf 时，从 seq_end>rcv_nxt 开始，一直读取到 seq>rcv_nxt 时停止。每读取一个包，需要更新 rcv_nxt。

```
void tcp_check_ofo_buf(struct tcp_sock *tsk)
{
  tcp_ofo_buf_entry_t *entry,*q;
  list_for_each_entry_safe(entry,q,&tsk->rcv_ofo_buf,list){
    if(less_or_equal_32b(entry->seq+entry->len,tsk->rcv_nxt)){
      list_delete_entry(&entry->list);
      free(entry->data);
      free(entry);
    }
    else if(greater_or_equal_32b(tsk->rcv_nxt,entry->seq)){
      int len = (int)(entry->seq+entry->len-tsk->rcv_nxt);
      write_ring_buffer(tsk->rcv_buf,entry->data+entry->len-len,len);
      tsk->rcv_nxt += len;
      list_delete_entry(&entry->list);
      free(entry->data);
      free(entry);
    }
    else break;
  }
}
```

（3）考虑连接建立/断开的情况

不需要为连接建立/断开实现额外的重传机制，上面的实现已经能够处理包括 SYN|FIN 的数据包。需要考虑的是状态切换：作为发送方，发出包后状态随即切换，重传时不会导致状态切换。作为接收方，收到包后需要判断包的 ack 等于 snd_nxt 才进行状态切换。这样，如果收到重复的 ACK，不会造成错误的切换。另外，在 CLOSE_WAIT 状态，需要等到发送的数据对方都接收了才切换状态，判断条件是 snd_nxt = snd_una。

4、实验结果

网络拓扑如图 1。生成的文件大小为 4052632B。

（1）C 程序发送，C 程序接收

shell 输出结果如图 2。由 diff 命令输出空结果，我的程序结果正确。

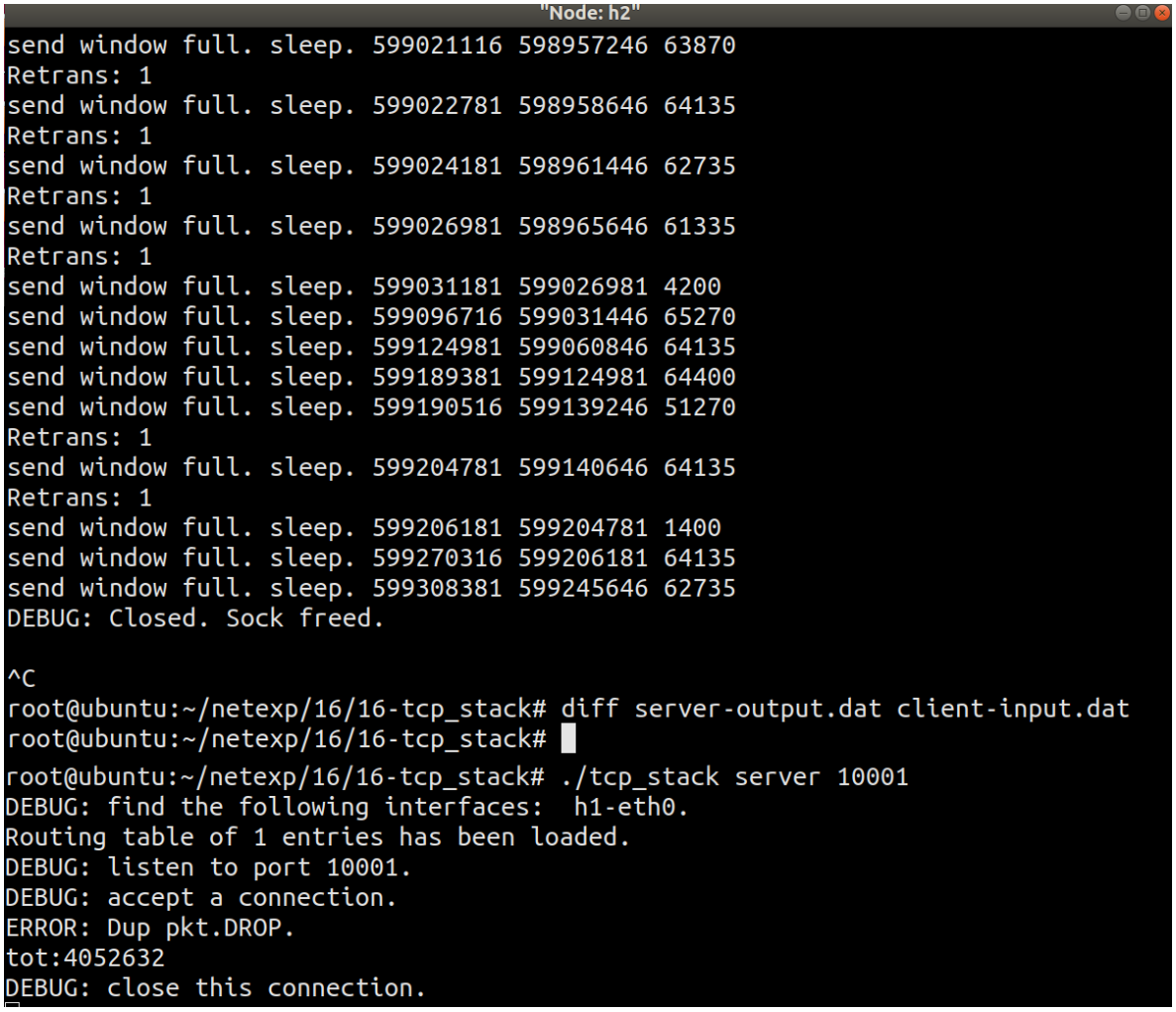

图 2

（2）Python 程序发送，C 程序接收

抓包结果如图 3，可以看到 FIN 包进行了重传，连接最后正确关闭。

实验结果如图 4，结果正确。

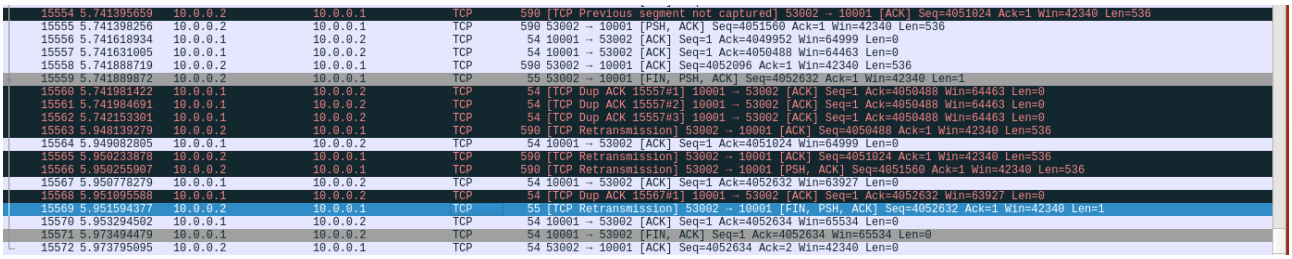

图 3

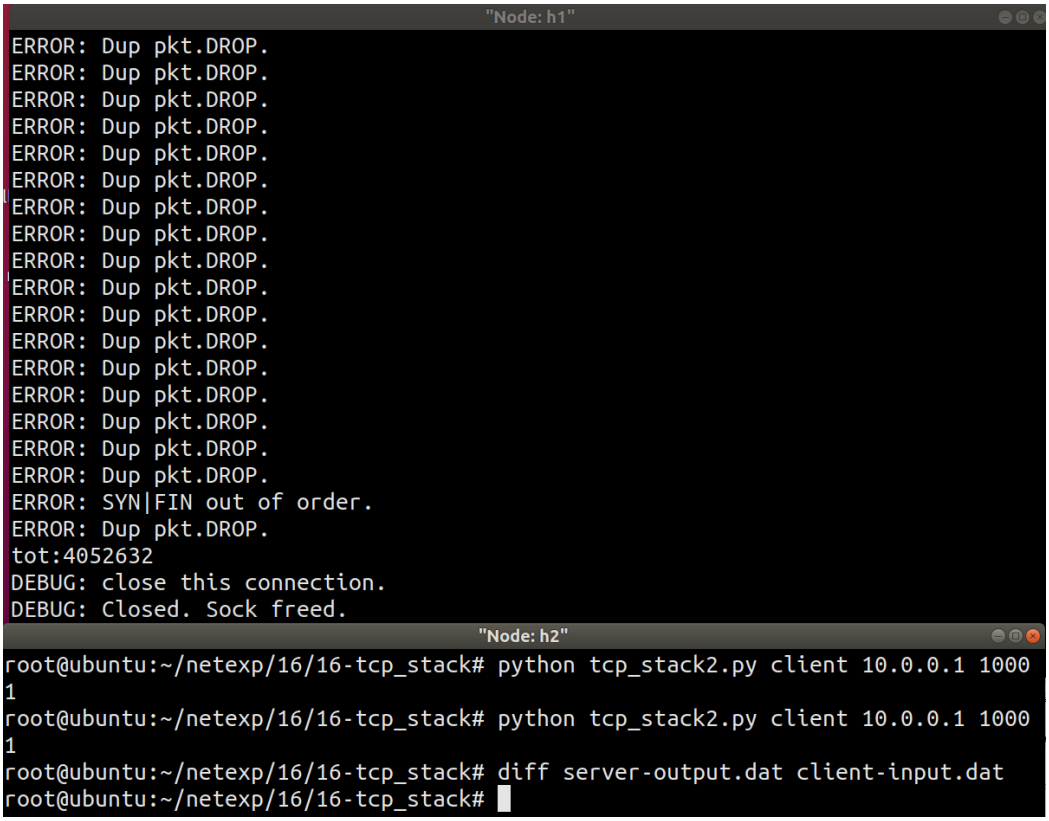

图 4

（3）C 程序发送，Python 程序接收

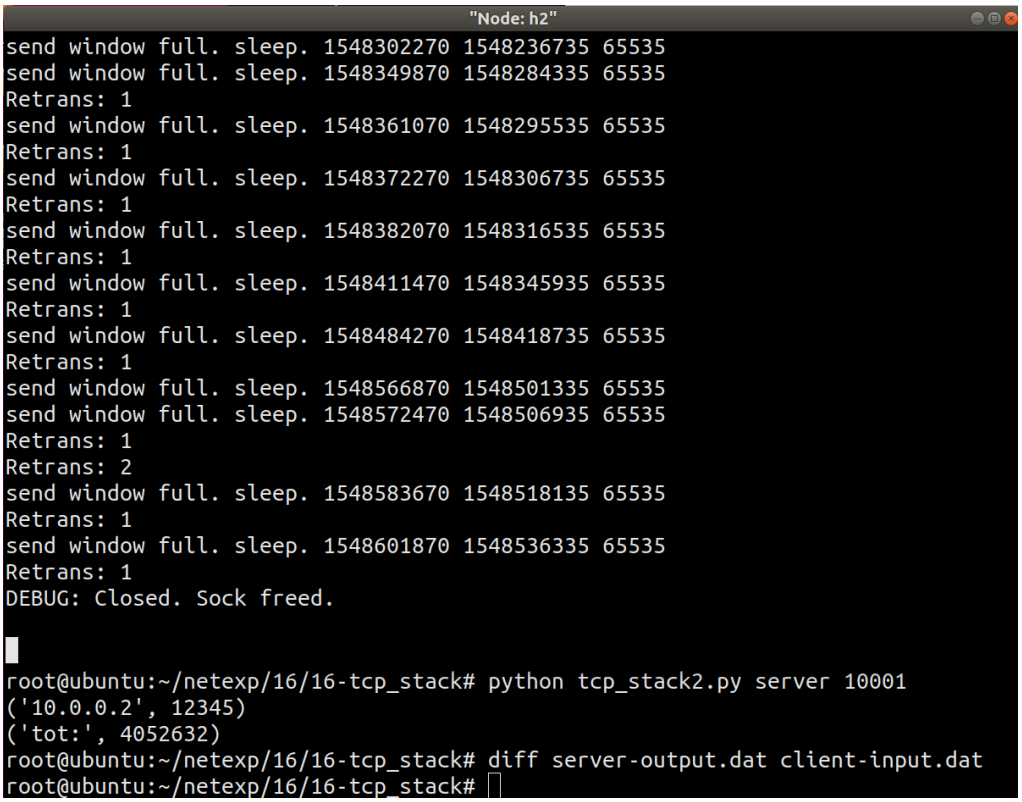

图 5

实验结果如图 5，结果正确。

每次重传一个包时，传输 4MB 数据需要十几秒。若改为每次重传发送队列中的所有包，则传输时间仅需不到 5 秒。

5、遇到的问题

（1）处理含 FIN 的乱序包

为了简化问题，不把它加入乱序队列，而是直接丢弃。