网络实验报告

雷正宇 leizhengyu16@mails.ucas.ac.cn

实验内容

实现 TCP 管理相关函数：

struct tcp_sock *alloc_tcp_sock();

int tcp_sock_bind(struct tcp_sock *, struct sock_addr *);

int tcp_sock_listen(struct tcp_sock *, int);

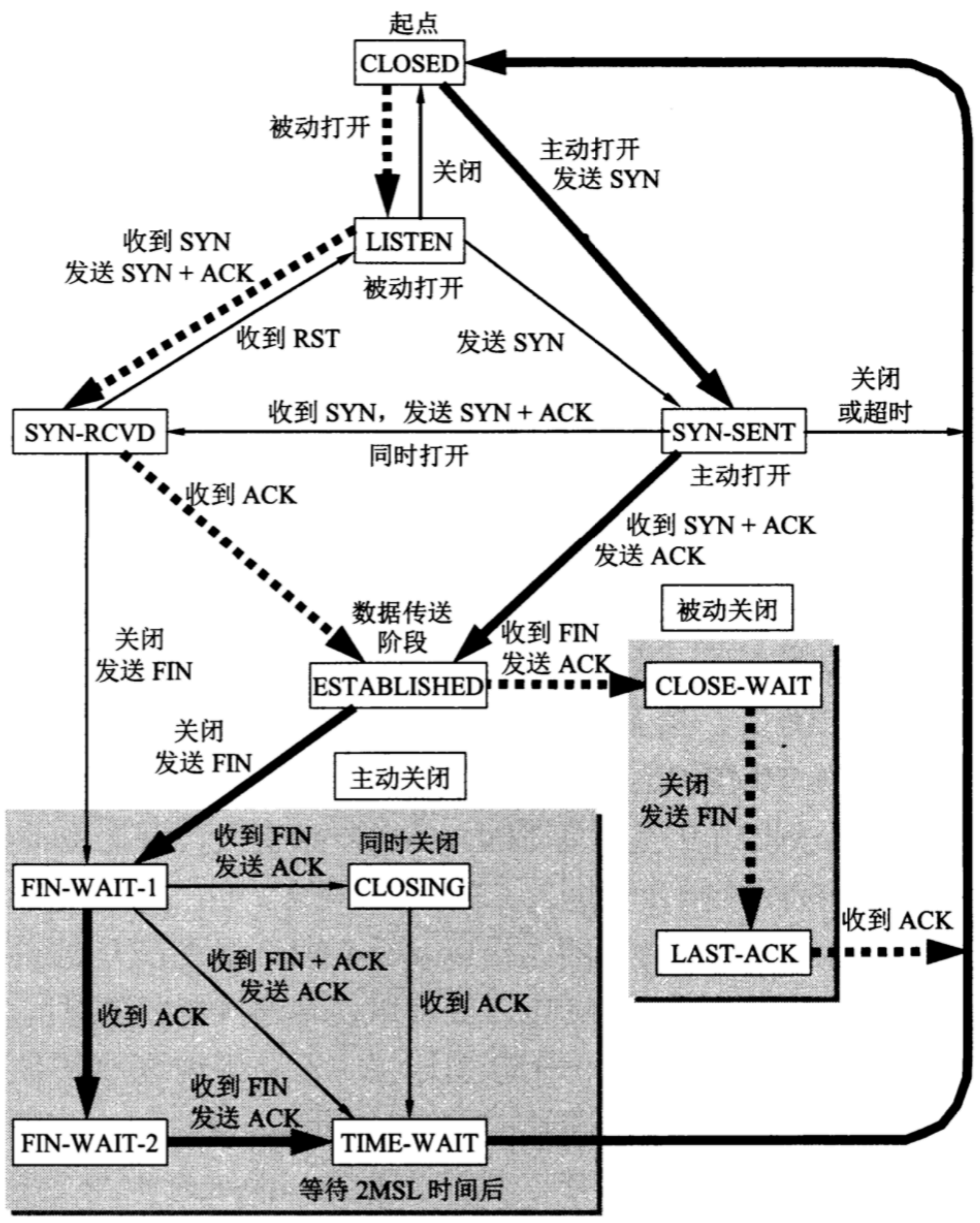
int tcp_sock_connect(struct tcp_sock *, struct sock_addr *);

struct tcp_sock *tcp_sock_accept(struct tcp_sock *);

void tcp_sock_close(struct tcp_sock *);

实验流程

本次试验要完成一个具备建立链接和断开链接功能的 TCP 协议栈。首先要明确 TCP 状态机的运作方式：



其中每一个箭头过程都需要处理。

处理控制信号过程

```
void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
  // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
  u8 flags = cb->flags;

  if (flags & TCP_RST) {
    tcp_sock_close(tsk);
    free_tcp_sock(tsk);
    return;
```

```c
  }

  log(DEBUG, IP_FMT ":%hu current state is %s.", \
      HOST_IP_FMT_STR(tsk->sk_sip), tsk->sk_sport, \
      tcp_state_str[tsk->state]);

  switch (tsk->state) {
    case TCP_CLOSED:
      tcp_send_reset(cb);
      return;
    case TCP_LISTEN:
      if (flags & TCP_SYN) {
        struct tcp_sock *new = alloc_tcp_sock();
        new->parent = tsk;
        new->sk_dip = cb->saddr;
        new->sk_dport = cb->sport;
        new->sk_sip = cb->daddr;
        new->sk_sport = cb->dport;

        new->iss = tcp_new_iss();
        new->snd_una = new->iss;
        new->snd_nxt = new->iss;
        new->rcv_nxt = cb->seq + 1;

        tcp_set_state(new, TCP_SYN_RECV);

        tcp_hash(new);
        list_add_head(&new->list, &tsk->listen_queue);

        tcp_send_control_packet(new, TCP_ACK | TCP_SYN);
      } else {
        log(ERROR, "invalid tcp packet while in state LISTEN");
      }
      return;
    case TCP_SYN_RECV:
      if (flags & TCP_ACK) {
        list_delete_entry(&tsk->list);
        tcp_sock_accept_enqueue(tsk);
        wake_up(tsk->parent->wait_accept);
        tcp_set_state(tsk, TCP_ESTABLISHED);
      } else {
        log(ERROR, "invalid tcp packet while in state SYN_RECV");
      }
      return;
    case TCP_SYN_SENT:
      if (flags & (TCP_SYN | TCP_ACK)) {
        tsk->rcv_nxt = cb->seq + 1;
        tsk->snd_nxt = cb->ack;

        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_state(tsk, TCP_ESTABLISHED);
        wake_up(tsk->wait_connect);
      } else if (flags & TCP_SYN) {
        tcp_send_control_packet(tsk, TCP_SYN | TCP_ACK);
        tcp_set_state(tsk, TCP_SYN_RECV);
      } else {
        log(ERROR, "invalid tcp packet while in state SYN_SENT");
      }
      return;
    case TCP_ESTABLISHED:
      if (flags & TCP_FIN) {
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_state(tsk, TCP_CLOSE_WAIT);
      } else {
        log(ERROR, "invalid tcp packet while in state ESTABLISHED");
      }
      return;
    case TCP_CLOSE_WAIT:
      log(ERROR, "invalid tcp packet while in state CLOSE_WAIT");
      return;
    case TCP_LAST_ACK:
      if (flags & TCP_ACK) {
        tcp_set_state(tsk, TCP_CLOSED);
        tcp_unhash(tsk);
      } else {
        log(ERROR, "invalid tcp packet while in state LAST_ACK");
      }
      return;
    case TCP_FIN_WAIT_1:
      if (flags & TCP_ACK) {
        tcp_set_state(tsk, TCP_FIN_WAIT_2);
      } else if (flags & TCP_FIN) {
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_state(tsk, TCP_CLOSING);
      } else {
        log(ERROR, "invalid tcp packet while in state FIN_WAIT_1");
      }
      return;
    case TCP_FIN_WAIT_2:
      if (flags & TCP_FIN) {
        tsk->rcv_nxt = cb->seq_end;
        tcp_send_control_packet(tsk, TCP_ACK);
        tcp_set_state(tsk, TCP_TIME_WAIT);
        tcp_set_timewait_timer(tsk);
      } else {
        log(ERROR, "invalid tcp packet while in state FIN_WAIT_2");
      }
    case TCP_CLOSING:
      if (flags & TCP_ACK) {
        tcp_set_state(tsk, TCP_TIME_WAIT);
        tcp_set_timewait_timer(tsk);
      } else {
        log(ERROR, "invalid tcp packet while in state FIN_WAIT_2");
      }
      return;
    case TCP_TIME_WAIT:
      log(ERROR, "invalid tcp packet while in state TIME_WAIT");
      return;
    default:
      break;
    }
}
```

该函数篇幅很长，但逻辑很清晰——先判断 TCP 目前的状态，再根据收到的控制信号进行相应的动作。其中部分动作只是改变此时的状态和发送对应控制信号，而少数动作则需要复杂操作。

当服务器进入 LISTEN 状态后，如果收到 SYN 信号，则需要新建一个子套接字，将其状态置为 SYN_RECV 并加入自己的监听队列。对于已经处于 SYN_RECV 状态的套接字，收到 ACK 包后要被唤醒。关于睡眠与唤醒的内容，下面还有叙述。接下来的几个过程都将和该过程有关联。

主动建立连接过程

```
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
  // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
  tsk->sk_dip = sock_addr_get_ip(skaddr);
  tsk->sk_dport = sock_addr_get_port(skaddr);
  iface_info_t *iface = list_entry(instance->iface_list.next, iface_info_t, list);
  tsk->sk_sip = iface->ip;
  tsk->sk_sport = tcp_get_port();

  tcp_bind_hash(tsk);

  tcp_send_control_packet(tsk, TCP_SYN);
  tcp_set_state(tsk, TCP_SYN_SENT);
  tcp_hash(tsk); // sequence?

  sleep_on(tsk->wait_connect);

  return 0;
}
```

过程分为4步：设置源/目的-端口/地址四元组，绑定到 bind_table，发送控制信号并切换状态，阻塞（睡眠）到 wait_connect 队列上。其中注意网络-本地字节序转换。等到收到 SYN + ACK 信号后，连接建立，该套接字进入 ESTABLISHED 状态。

被动监听建立过程

```c
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
  // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
  tsk->backlog = backlog;
  tcp_set_state(tsk, TCP_LISTEN);
  return tcp_hash(tsk);
}
```

该过程也很简单，修改套接字状态并将套接字塞入哈希表即可。

接收过程

```c
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
  // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
  while (list_empty(&tsk->accept_queue)) {
    sleep_on(tsk->wait_accept);
  }

  return tcp_sock_accept_dequeue(tsk);
}
```

一个被动建立连接方处理过程是这样的：从主套接字的 accept_queue 中获取队列头的子套接字，然后处理该子套接字。一般来说，以 server 为例（本实验不是这样的例子），一个 server 在建立主套接字后会不断针对主套接字的 accept_queue 进行处理，而封装调用即是 tcp_sock_accept. 回到状态机处理信号的过程中，在本实验中，暂时不处理读写过程，所以当被动建立连接的一方处于 SYN_RECV 状态时，收到来自主动建立连接一方的 ACK 信号后，会唤醒 wait_accept 上的子套接字。

关闭过程

```c
void tcp_sock_close(struct tcp_sock *tsk)
{
  // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
  switch(tsk->state) {
    case TCP_CLOSED:
      break;
    case TCP_LISTEN:
      tcp_unhash(tsk);
      tcp_bind_unhash(tsk);
      tcp_set_state(tsk, TCP_CLOSED);
      break;
    case TCP_SYN_SENT:
      tcp_unhash(tsk);
      tcp_set_state(tsk, TCP_CLOSED);
      break;
  case TCP_ESTABLISHED:
   tcp_set_state(tsk, TCP_FIN_WAIT_1);
   tcp_send_control_packet(tsk, TCP_FIN);
   break;
  case TCP_CLOSE_WAIT:
   tcp_set_state(tsk, TCP_LAST_ACK);
   tcp_send_control_packet(tsk, TCP_FIN);
   break;
  default:
   break; // temp deal
 }
}
```

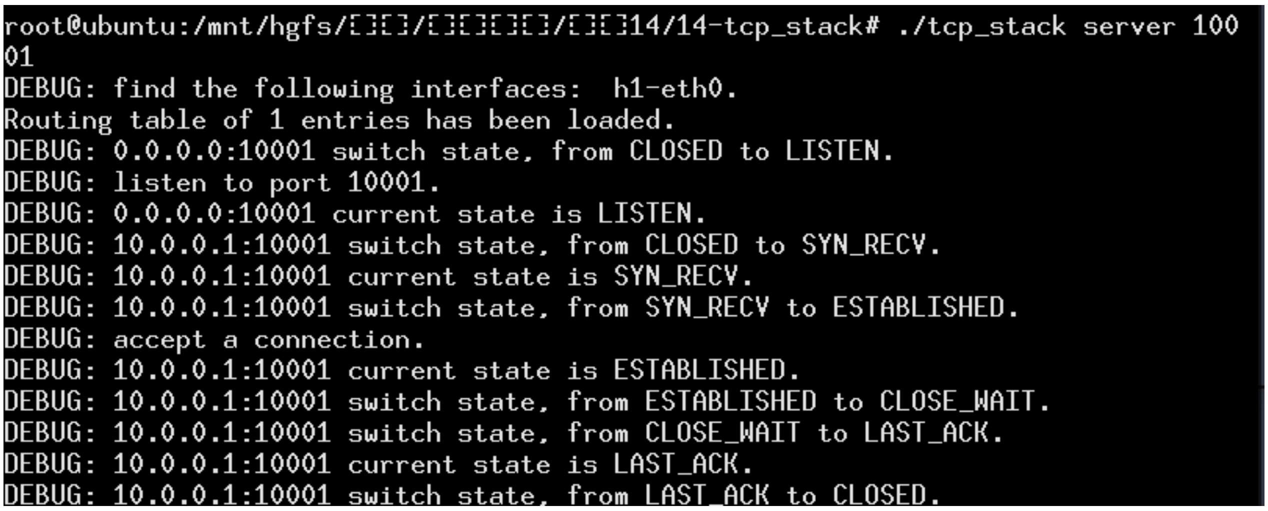
要考虑在任何场景下某一方选择 close 整个套接字，因为这是 TCP 应用用来彻底释放资源的唯一调用。由于本例比较简单，而且数据传输不会出错，所以不用考虑得太多，关注资源的释放即可。

实验结果与分析

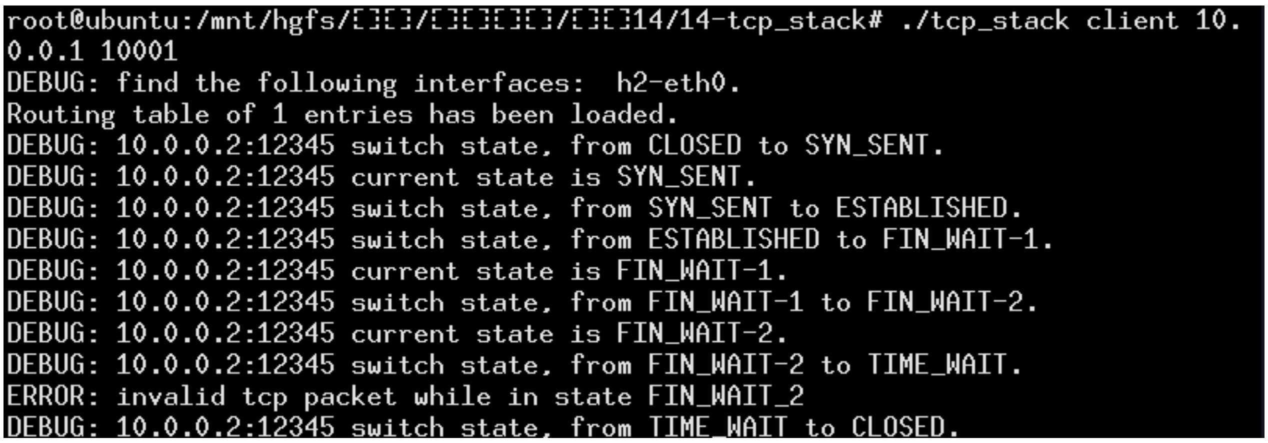
我在实验代码中加入了一些 DEBUG 信息用来检测套接字的状态。利用实验给定的 server 和 client 小应用，可以看到两边套接字的状态切换过程，从而验证 TCP 实现的正确性：

```
运行给定网络拓扑(tcp_topo.py)
在节点h1上执行TCP程序
执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
在h1上运行TCP协议栈的服务器模式  (./tcp_stack server 10001)
在节点h2上执行TCP程序
执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能
在h2上运行TCP协议栈的客户端模式，连接至h1，显示建立连接成功后自动关闭连接 (./tcp_stack client 10.0.0.1 10001)
```

结果：



```
root@ubuntu:/mnt/hgfs/□□/□□□□/□□14/14-tcp_stack# ./tcp_stack server 100
01
DEBUG: find the following interfaces:  h1-eth0.
Routing table of 1 entries has been loaded.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: listen to port 10001.
DEBUG: 0.0.0.0:10001 current state is LISTEN.
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: 10.0.0.1:10001 current state is SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
DEBUG: accept a connection.
DEBUG: 10.0.0.1:10001 current state is ESTABLISHED.
DEBUG: 10.0.0.1:10001 switch state, from ESTABLISHED to CLOSE_WAIT.
DEBUG: 10.0.0.1:10001 switch state, from CLOSE_WAIT to LAST_ACK.
DEBUG: 10.0.0.1:10001 current state is LAST_ACK.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
```



```
root@ubuntu:/mnt/hgfs/[][]/[][][][]/[][]14/14-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: 10.0.0.2:12345 switch state, from CLOSED to SYN_SENT.
DEBUG: 10.0.0.2:12345 current state is SYN_SENT.
DEBUG: 10.0.0.2:12345 switch state, from SYN_SENT to ESTABLISHED.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 current state is FIN_WAIT-1.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 current state is FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
ERROR: invalid tcp packet while in state FIN_WAIT_2
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```