网络传输机制实验一报告

热伊莱·图尔贡 2018K8009929030

一、实验题目：网络传输机制实验一

二、实验内容

内容一：连接管理

- 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断开连接 (./tcp_stack client 10.0.0.1 10001)
- 可以在一端用tcp_stack_conn.py替换tcp_stack执行，测试另一端
- 通过wireshark抓包来来验证建立和断开连接的正确性

内容二：短消息收发

- 参照tcp_stack_trans.py，修改tcp_apps.c，使之能够收发短消息

- 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_offloading.sh , disable_tcp_rst.sh)
  - 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_offloading.sh, disable_tcp_rst.sh)
  - 在h2上运行TCP协议栈的客户端模式，连接h1并正确收发数据 (./tcp_stack client 10.0.0.1 10001)
    - client向server发送数据，server将数据echo给client
- 使用tcp_stack_trans.py替换其中任意一端，对端都能正确收发数据

内容三：大文件传送

- 修改tcp_apps.c(以及tcp_stack_trans.py)，使之能够收发文件
- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 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_offloading.sh , disable_tcp_rst.sh)
  - 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_offloading.sh, disable_tcp_rst.sh)
  - 在h2上运行TCP协议栈的客户端模式 (./tcp_stack client 10.0.0.1 10001)
    - Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat
- 使用md5sum比较两个文件是否完全相同
- 使用tcp_stack_trans.py替换其中任意一端，对端都能正确收发数据

三、实验过程

1.连接管理

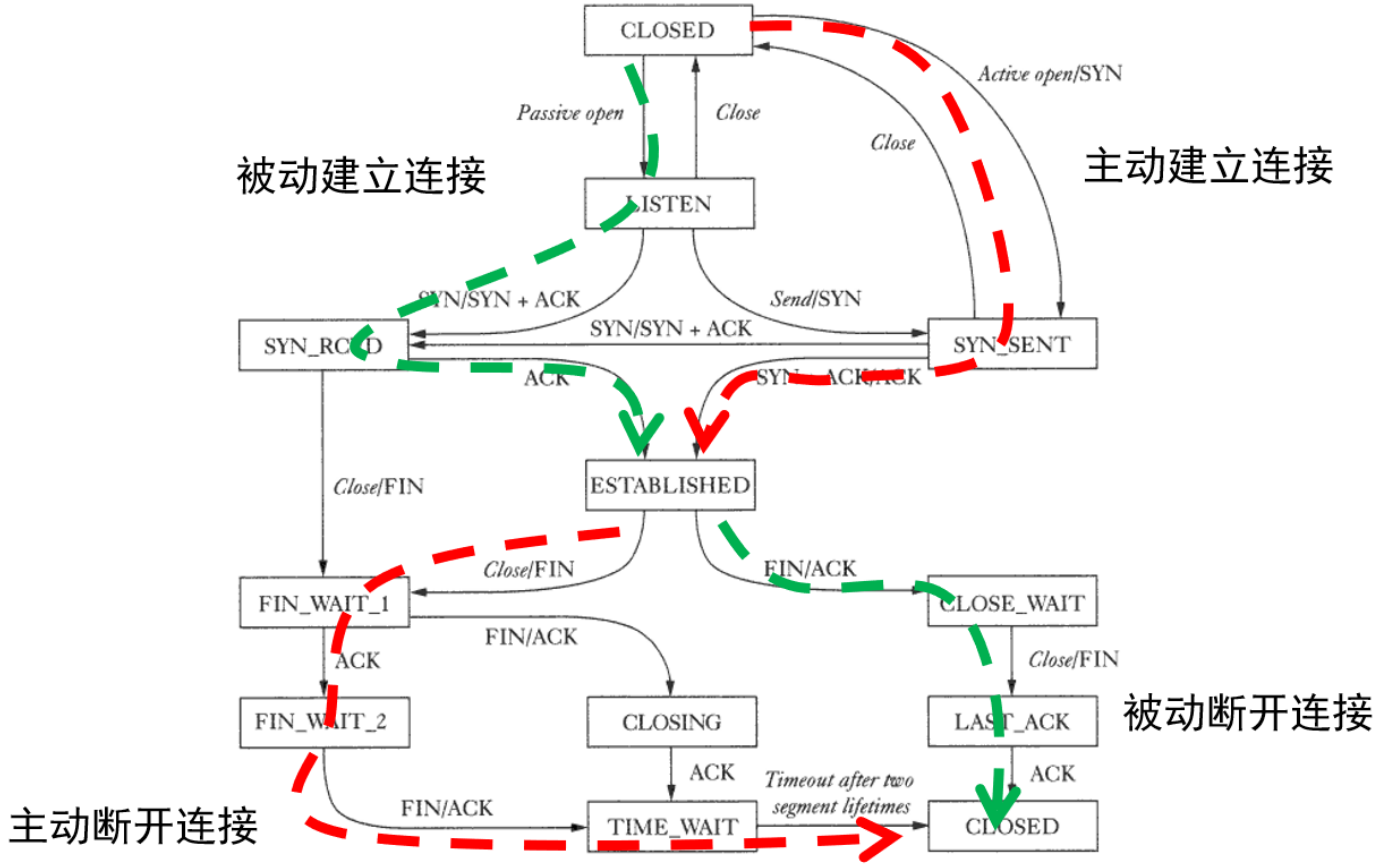
```c
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
        // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

        tsk->backlog = backlog;
        tcp_set_state(tsk, TCP_LISTEN);
        tcp_hash(tsk);

        // return -1;
        // return 0;
}
```

```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);
        }
        struct tcp_sock *pop = tcp_sock_accept_dequeue(tsk);
        tcp_set_state(pop, TCP_ESTABLISHED);
        tcp_hash(pop);

        // return NULL;
        return pop;
}
```

```c
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
        // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

        u16 sport = tcp_get_port();
        if (sport == 0)
        {
                return -1;
        }
        u32 saddr = longest_prefix_match(ntohl(skaddr->ip))->iface->ip;
        tsk->sk_sip = saddr;
        tsk->sk_sport = sport;
        tsk->sk_dip = ntohl(skaddr->ip);
        tsk->sk_dport = ntohs(skaddr->port);

        tcp_bind_hash(tsk);

        tcp_send_control_packet(tsk, TCP_SYN);
        tcp_set_state(tsk, TCP_SYN_SENT);
        tcp_hash(tsk);
        sleep_on(tsk->wait_connect);

        // return -1;
        return sport;
}
```

2.状态机转移

TCP连接管理和状态迁移



只实现虚线标识的路径过程

```c
// Process the incoming packet according to TCP state machine.
void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)
{
	// fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

	if (tsk == NULL)
	{
		tcp_send_reset(cb);
		return;
	}
	if (cb->flags & TCP_RST)
	{
		tcp_set_state(tsk, TCP_CLOSED);
		tcp_unhash(tsk);
		tcp_bind_unhash(tsk);
		return;
	}

	if (tsk->state == TCP_LISTEN)
	{
		if (cb->flags == TCP_SYN)
		{
			struct tcp_sock *child_tsk = alloc_tcp_sock();
			child_tsk->parent = tsk;
			tsk->ref_cnt += 1;

			child_tsk->local.ip = cb->daddr;
			child_tsk->local.port = cb->dport;
			child_tsk->peer.ip = cb->saddr;
			child_tsk->peer.port = cb->sport;

			child_tsk->iss = tcp_new_iss();
			child_tsk->snd_nxt = child_tsk->iss;
			child_tsk->rcv_nxt = cb->seq_end;

			tcp_set_state(child_tsk, TCP_SYN_RECV);

			tcp_hash(child_tsk);
```

```
                init_list_head(&child_tsk->bind_hash_list);

                log(DEBUG, "Pass " IP_FMT ":%hu <-> " IP_FMT ":%hu from process to listen_queue",
                    HOST_IP_FMT_STR(child_tsk->sk_sip), child_tsk->sk_sport,
                    HOST_IP_FMT_STR(child_tsk->sk_dip), child_tsk->sk_dport);
                list_add_tail(&child_tsk->list, &tsk->listen_queue);

                tcp_send_control_packet(child_tsk, TCP_SYN | TCP_ACK);
            }
        }
        else if (tsk->state == TCP_SYN_RECV)
        {
            if (cb->flags == TCP_ACK)
            {
                if (!is_tcp_seq_valid(tsk, cb))
                {
                    return;
                }
                tsk->rcv_nxt = cb->seq_end;
                tsk->snd_una = cb->ack;

                if (tsk->parent)
                {
                    if (tcp_sock_accept_queue_full(tsk->parent))
                    {
                        tcp_set_state(tsk, TCP_CLOSED);
                        tcp_send_control_packet(tsk, TCP_RST);

                        tcp_unhash(tsk);
                        tcp_bind_unhash(tsk);
                        list_delete_entry(&tsk->list);
                        free_tcp_sock(tsk);
                    }
                    else
                    {
                        tcp_set_state(tsk, TCP_ESTABLISHED);
                        tcp_sock_accept_enqueue(tsk);
                        wake_up(tsk->parent->wait_accept);
                    }
                }
                else
                {
                    tcp_set_state(tsk, TCP_ESTABLISHED);
                    wake_up(tsk->parent->wait_connect);
                }
            }
        }
        else if (tsk->state == TCP_SYN_SENT)
        {
            if (cb->flags == (TCP_SYN | TCP_ACK))
            {
                tsk->rcv_nxt = cb->seq_end;
                tsk->snd_una = cb->ack;

                tcp_set_state(tsk, TCP_ESTABLISHED);
                tcp_send_control_packet(tsk, TCP_ACK);

                wake_up(tsk->wait_connect);
            }
            else if (cb->flags == TCP_SYN)
            {
                tsk->rcv_nxt = cb->seq_end;
                tcp_set_state(tsk, TCP_SYN_RECV);
                tcp_send_control_packet(tsk, TCP_SYN | TCP_ACK);
            }
        }
        else if (tsk->state == TCP_FIN_WAIT_1)
        {
            if (cb->flags == TCP_ACK)
            {
                if (!is_tcp_seq_valid(tsk, cb))
                {
                    return;
                }
                tsk->rcv_nxt = cb->seq_end;
                tsk->snd_una = cb->ack;

                tcp_set_state(tsk, TCP_FIN_WAIT_2);
            }
            else if (cb->flags == TCP_FIN)
            {
                if (!is_tcp_seq_valid(tsk, cb))
```

```
121                 if (!is_tcp_seq_valid(tsk, cb))
122                 {
123                     return;
124                 }
125                 tsk->rcv_nxt = cb->seq_end;
126                 tcp_set_state(tsk, TCP_CLOSING);
127                 tcp_send_control_packet(tsk, TCP_ACK);
128             }
129             else if (cb->flags == (TCP_FIN | TCP_ACK))
130             {
131                 if (!is_tcp_seq_valid(tsk, cb))
132                 {
133                     return;
134                 }
135                 tsk->rcv_nxt = cb->seq_end;
136                 tsk->snd_una = cb->ack;
137 
138                 tcp_set_state(tsk, TCP_TIME_WAIT);
139                 tcp_set_timewait_timer(tsk);
140                 tcp_send_control_packet(tsk, TCP_ACK);
141             }
142         }
143         else if (tsk->state == TCP_ESTABLISHED)
144         {
145             if (!is_tcp_seq_valid(tsk, cb))
146             {
147                 return;
148             }
149             tsk->rcv_nxt = cb->seq_end;
150 
151             if (cb->flags & TCP_ACK)
152             {
153                 tsk->snd_una = cb->ack;
154             }
155 
156             if (cb->flags & TCP_FIN)
157             {
158                 tsk->rcv_nxt = cb->seq_end;
159                 tcp_set_state(tsk, TCP_CLOSE_WAIT);
160                 tcp_send_control_packet(tsk, TCP_ACK);
161             }
162         }
163         else if (tsk->state == TCP_CLOSING)
164         {
165             if (cb->flags == TCP_ACK)
166             {
167                 if (!is_tcp_seq_valid(tsk, cb))
168                 {
169                     return;
170                 }
171                 tsk->rcv_nxt = cb->seq_end;
172                 tsk->snd_una = cb->ack;
173 
174                 tcp_set_state(tsk, TCP_TIME_WAIT);
175                 tcp_set_timewait_timer(tsk);
176             }
177         }
178         else if (tsk->state == TCP_TIME_WAIT)
179         {
180             // log(DEBUG, "TCP_TIME_WAIT");
181         }
182         else if (tsk->state == TCP_CLOSE_WAIT)
183         {
184             // log(DEBUG, "TCP_CLOSE_WAIT");
185         }
186         else if (tsk->state == TCP_LAST_ACK)
187         {
188             if (cb->flags == TCP_ACK)
189             {
190                 if (!is_tcp_seq_valid(tsk, cb))
191                 {
192                     return;
193                 }
194                 tsk->rcv_nxt = cb->seq_end;
195                 tsk->snd_una = cb->ack;
196 
197                 tcp_set_state(tsk, TCP_CLOSED);
198                 tcp_unhash(tsk);
199                 tcp_bind_unhash(tsk);
200             }
201         }
202         else if (tsk->state == TCP_CLOSED)
203         {
```

```
        tcp_unhash(tsk);
        tcp_bind_unhash(tsk);
    }
}
```

3.TCP计时器管理

```
// scan the timer_list, find the tcp sock which stays for at 2*MSL, release it
void tcp_scan_timer_list()
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);

    struct tcp_timer *time_entry, *time_q;
    list_for_each_entry_safe(time_entry, time_q, &timer_list, list)
    {
        if (time_entry->enable == 1 && (time(NULL) * TCP_MSL - time_entry->timeout * TCP_MSL > TCP_TIMEWAIT_TIMEOUT))
        {
            struct tcp_sock *tsk = timewait_to_tcp_sock(time_entry);
            list_delete_entry(&time_entry->list);
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_bind_unhash(tsk);
        }
    }
}
```

```
// set the timewait timer of a tcp sock, by adding the timer into timer_list
void tcp_set_timewait_timer(struct tcp_sock *tsk)
{
    // fprintf(stdout, "TODO: implement %s please.\n", __FUNCTION__);
    // tsk->timewait.enable = 1;
    tsk->timewait.type = 0;
    tsk->timewait.timeout = TCP_TIMEWAIT_TIMEOUT;
    list_add_tail(&tsk->timewait.list, &timer_list);
    tsk->ref_cnt++;
}
```

四、实验结果

由于实验环境出问题，未能进行验证。