

实验报告

网络传输机制实验一

一、实验内容

了解 Socket 数据结构、TCP 连接管理和状态转移以及数据包处理流程，实现 TCP 协议栈的相应服务函数，实现 TCP 建立连接与断开连接的数据包处理机制。

二、实验流程

1. 实现 TCP 协议栈相关操作。
2. 实现 TCP 连接和数据包处理机制。
3. 在给定拓扑下验证相应功能的正确性。

三、实验结果及分析

（一）TCP 功能实现思路

1、TCP 协议栈

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
```

函数，设置 backlog，将状态切换到 LISTEN，并将 socket 加入 listen hash 表。

```
int tcp_sock_listen(struct tcp_sock *tsk, int backlog)
{
    tsk->backlog = backlog;
    tcp_set_state(tsk, TCP_LISTEN);
    return tcp_hash(tsk);
}
```

```
struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
```

函数，作为服务器的一方等待连接，如果没有成功连接的 socket，即 accept_queue 为空，阻塞等待。当被唤醒后，从 accept_queue 获取成功建立连接的 child socket，将其状态转换为 ESTABLISHED，并加入到 established hash 表。最后返回 child socket。

```

struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)
{
    while (list_empty(&tsk->accept_queue)) {
        sleep_on(tsk->wait_accept);
    }

    struct tcp_sock *child;
    if ((child = tcp_sock_accept_dequeue(tsk)) != NULL) {
        tcp_set_state(child, TCP_ESTABLISHED);
        if (tcp_hash(child) == 0)
            return child;
        else
            return NULL;
    }

    return NULL;
}

```

```
int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
```

函数，作为客户端的一方主动发起连接，此时可以完全确定四元组。首先分配源端口，查找源 IP 地址，确定四元组。然后发出 SYN 数据包，请求连接。将状态转为 SYN_SENT，把 socket 加入 established hash 表。最后 sleep on，等待客户端回应。

```

int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)
{
    u16 sport = tcp_get_port();
    if (sport == 0) {
        return -1;
    }
    rt_entry_t *entry = longest_prefix_match(ntohl(skaddr->ip));
    if (entry == NULL) {
        return -1;
    }
    tsk->sk_sip = entry->iface->ip;
    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 sport;
}

```

```
void tcp_sock_close(struct tcp_sock *tsk)
```

函数，完成主动断开和被动断开连接，因此分两种情况处理。当前为 ESTABLISHED 状态，此时为主动断开连接，向对方发送 FIN 和 ACK 信号，并转到 FIN_WAIT_1 状态。当前为 CLOSE_WAIT 状态，此时为被动方断开连接，向对方发送 FIN 和 ACK 信号，并转到 LAST_ACK 状态。最后如果为其他状态，直接断开连接，释放资源。

```

void tcp_sock_close(struct tcp_sock *tsk)
{
    switch (tsk->state) {
        case TCP_ESTABLISHED: {
            tcp_set_state(tsk, TCP_FIN_WAIT_1);
            tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
            break;
        }
        case TCP_CLOSE_WAIT: {
            tcp_set_state(tsk, TCP_LAST_ACK);
            tcp_send_control_packet(tsk, TCP_FIN|TCP_ACK);
            break;
        }
        default: {
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
            break;
        }
    }
}

```

2、TCP 连接管理和数据包处理

`struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)`

函数，查找 listen hash 表。注意只用 sport 作为 key，不使用 saddr，因为此时以 0.0.0.0 代表主机自身。

```

struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)
{
    int hash = tcp_hash_function(0, 0, sport, 0);
    struct list_head *list = &tcp_listen_sock_table[hash];

    struct tcp_sock *tmp;
    list_for_each_entry(tmp, list, hash_list) {
        if (sport == tmp->sk_sport)
            return tmp;
    }

    return NULL;
}

```

`struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport)`

函数，查找 established hash 表。与查找 listen hash 表基本相同，此时使用 4 元组作为 key。

`void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)`

函数，处理 TCP 数据包和连接管理。

检查校验和以及 socket 是否存在放在上一级函数中处理，这里略过。

首先处理 RST 包，如果为 RST，直接结束连接，回收资源。

然后是对建立连接进行处理。分为 3 种状态：LISTEN 状态收到 SYN，回应建立连接，发送 SYN 和 ACK 包，同时建立一个 child socket 来负责与该客户进行连接。SYN_SENT 状态，收到 SYN 或者 ACK，说明服务器响应握手，状态转换为 ESTABLISHED，更新 rcv_nxt、snd_una，发送 ACK 包回应，最后唤醒客户端进程。SYN_RECV 状态，收到 ACK，3 次握手完成。如果 accept queue 未满，加入 accept queue，唤醒服务器进程。

```
switch (tsk->state) {
    case TCP_LISTEN: {
        if (tcp->flags & TCP_SYN) {
            tcp_set_state(tsk, TCP_SYN_RECV);
            struct tcp_sock *child = alloc_child_tcp_sock(tsk, cb);
            tcp_send_control_packet(child, TCP_SYN|TCP_ACK);
        }
        return;
    }
    case TCP_SYN_SENT: {
        if (tcp->flags & (TCP_ACK | TCP_SYN)) {
            tcp_set_state(tsk, TCP_ESTABLISHED);
            tsk->rcv_nxt = cb->seq + 1;
            tsk->snd_una = cb->ack;
            wake_up(tsk->wait_connect);
            tcp_send_control_packet(tsk, TCP_ACK);
        }
        return;
    }
    case TCP_SYN_RECV: {
        if (tcp->flags & TCP_ACK) {
            if (tcp_sock_accept_queue_full(tsk)) {
                return;
            }
            struct tcp_sock *csk = tcp_sock_listen_dequeue(tsk);
            tcp_sock_accept_enqueue(csk);
            //tcp_set_state(csk, TCP_ESTABLISHED);
            csk->rcv_nxt = cb->seq;
            csk->snd_una = cb->ack;
            wake_up(tsk->wait_accept);
        }
        return;
    }
    default: {
        break;
    }
}
```

处理完 SYN 包，检查 ACK 字段。

```
if (!is_tcp_seq_valid(tsk, cb)) {
    return;
}
```

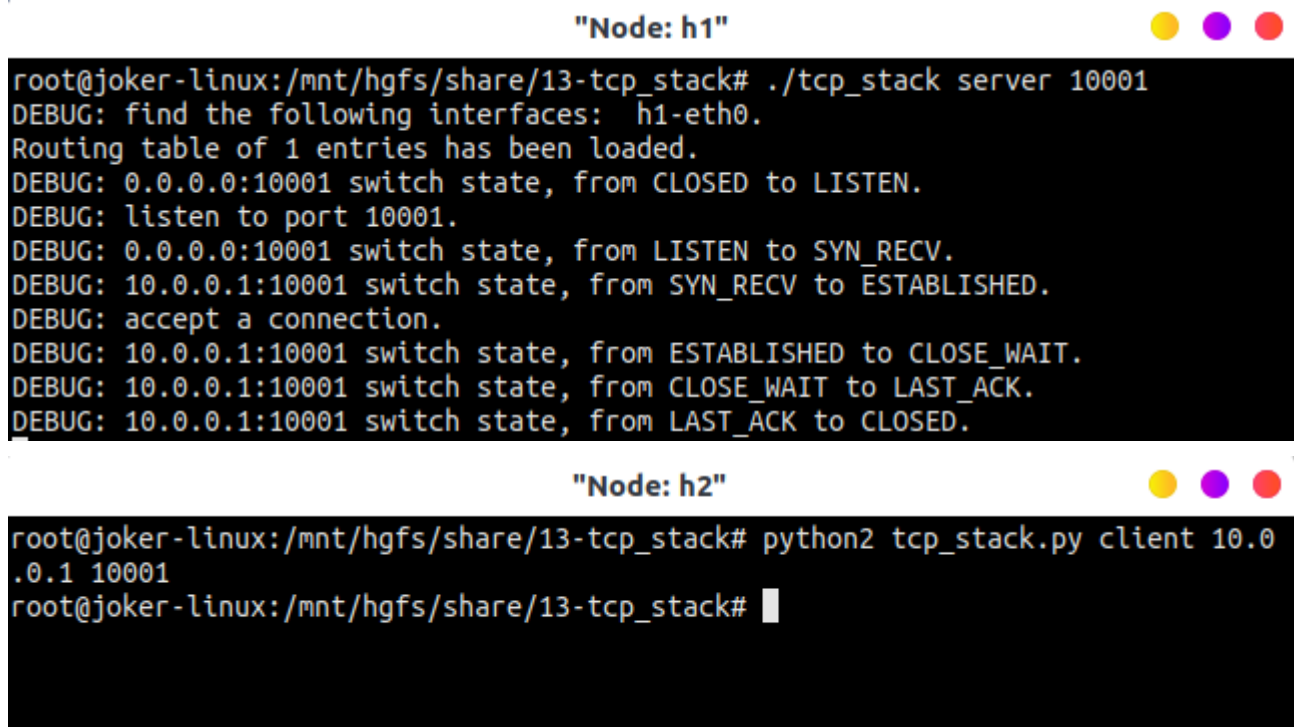
最后处理断开连接与 FIN 包。分为四种状态分别处理，这里不再详细赘述。注意两点，一是 ACK 不消耗序号，而 FIN 要消耗序号；二是主动断开连接的一方转到 TIME_WAIT 状态后，不立刻断开，需要设置定时器，等到时间后由另一线程正式断开连接并释放资源。

（二）实验验证功能

1、本实验 Server 与标准 Client

H1: 本实验 server

H2: 标准 client



The image shows two terminal windows. The top window, titled '"Node: h1"', shows the execution of the tcp_stack server. The bottom window, titled '"Node: h2"', shows the execution of the tcp_stack client.

```
"Node: h1"
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# ./tcp_stack server 10001
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 switch state, from LISTEN to 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 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 switch state, from LAST_ACK to CLOSED.

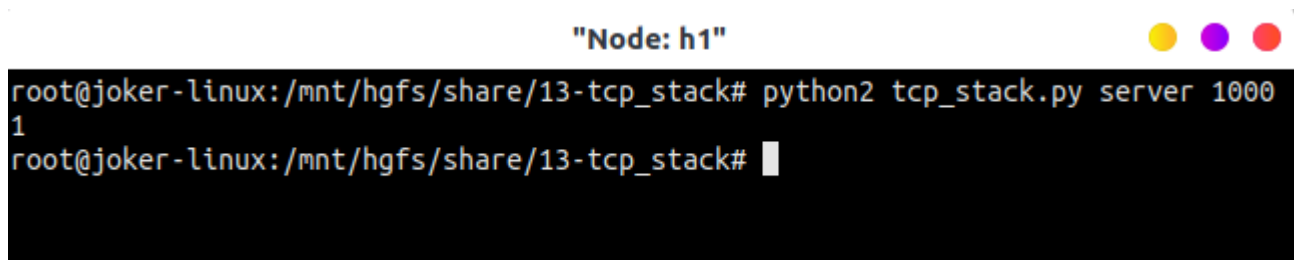
"Node: h2"
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# python2 tcp_stack.py client 10.0
.0.1 10001
root@joker-linux:/mnt/hgfs/share/13-tcp_stack#
```

H1 状态变化符合预期结果。

2、标准 Server 与本实验 Client

H1: 标准 server

H2: 本实验 client



The image shows a terminal window titled '"Node: h1"' where the standard server is running.

```
"Node: h1"
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# python2 tcp_stack.py server 1000
1
root@joker-linux:/mnt/hgfs/share/13-tcp_stack#
```

"Node: h2"

```
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# ./tcp_stack client 10.0.0.1 1000
1
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 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 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

H2 状态变化符合预期结果。

3、本实验 Server 与本实验 Client

H1: 本实验 server

H2: 本实验 client

"Node: h1"

```
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# ./tcp_stack server 10001
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 switch state, from LISTEN to 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 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 switch state, from LAST_ACK to CLOSED.
```

"Node: h2"

```
root@joker-linux:/mnt/hgfs/share/13-tcp_stack# ./tcp_stack client 10.0.0.1 1000
1
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 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 switch state, from FIN_WAIT-1 to FIN_WAIT-2.
DEBUG: 10.0.0.2:12345 switch state, from FIN_WAIT-2 to TIME_WAIT.
DEBUG: 10.0.0.2:12345 switch state, from TIME_WAIT to CLOSED.
```

H1、H2 行为与前面与标准程序交互时一致，符合预期结果。

4、Wireshark 抓包验证

标准 server 与标准 client 的抓包结果如下：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	72:7a:36:36:c7:41	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.011946896	56:8b:96:19:b4:71	72:7a:36:36:c7:41	ARP	42	10.0.0.1 is at 56:8b:96:19:b4:71
3	0.023830889	10.0.0.2	10.0.0.1	TCP	74	47578 → 10001 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=4168892016 TSecr=1504356321
4	0.036407939	10.0.0.1	10.0.0.2	TCP	74	10001 → 47578 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=1504356321 TSecr=4168892016
5	0.048904517	10.0.0.2	10.0.0.1	TCP	66	47578 → 10001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=4168892063 TSecr=1504356321
6	1.049659829	10.0.0.2	10.0.0.1	TCP	66	47578 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=4168893065 TSecr=1504356321
7	1.065392548	10.0.0.1	10.0.0.2	TCP	66	10001 → 47578 [ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=1504357351 TSecr=4168893065
8	5.083734182	10.0.0.1	10.0.0.2	TCP	66	10001 → 47578 [FIN, ACK] Seq=1 Ack=2 Win=43520 Len=0 TSval=1504361365 TSecr=4168893065
9	5.099117855	10.0.0.2	10.0.0.1	TCP	66	47578 → 10001 [ACK] Seq=2 Ack=2 Win=42496 Len=0 TSval=4168897110 TSecr=1504361365

本实验 server 与本实验 client 的抓包结果如下：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	da:51:9a:ad:ed:7e	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.011654901	8a:65:7e:9a:93:7e	da:51:9a:ad:ed:7e	ARP	42	10.0.0.1 is at 8a:65:7e:9a:93:7e
3	0.011675776	8a:65:7e:9a:93:7e	da:51:9a:ad:ed:7e	ARP	42	10.0.0.1 is at 8a:65:7e:9a:93:7e
4	0.024387894	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.034993010	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.047608466	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.047613400	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.058828754	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.076134136	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.090880681	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

对比可知，结果一致，本实验实现的功能正确。

四、实验总结

通过本次实验，我了解了 TCP 协议栈的功能、socket 数据结构的设计，以及 TCP 建立和断开连接的处理流程，这让我对 TCP 协议有了基本的了解，为之后的实验奠定了基础。