Tcp-stack 实验设计报告

中国科学院大学
页
2022 年 5 月 28 日

一、网络传输机制实验一

1. 实验内容

(1) 基于已有代码框架，理解 TCP 连接流程和 Socket 数据结构，完成 TCP Sock 的连接管理相关函数和 TCP 协议栈建立连接与断开连接的数据包处理机制，实现简单的 TCP 连接。

(2) 在上面的基础上实现数据传输，利用环形缓存 Receiving Buffer 完成数据的接收和缓存，使得结点之间能在无丢包网络环境中传输数据，完成字符串回显和大文件传输的任务。其次，实现流量控制：通过调整 recv_window 来表达自己的接收能力。

2. 实验流程

(一) 实验内容 1：连接管理

①完成代码编写后编译；

②在如图 1 所示拓扑的节点 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；

③在一端用 tcp_stack_conn.py 替换 tcp_stack 执行，测试另一端，验证是否能和 linux 协议栈正常建立连接；

④通过 wireshark 抓包测试正确性.

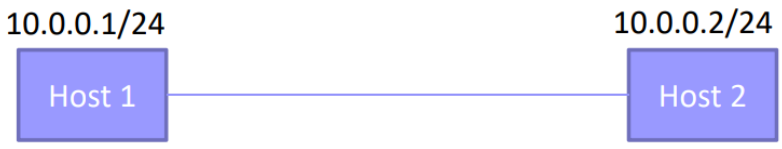


图 1

(二)实验内容 2:数据传输

①参照 tcp_stack_trans.py，修改 tcp_apps.c，使之能够收发短消息；

②在如图 1 所示拓扑的节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送参照 tcp_stack_trans.py，修改 tcp_apps.c，使之能够收发短消息；
③在一端用 tcp_stack_trans.py 替换 tcp_stack 执行，测试另一端，验证是否能和 linux 协议栈正常建立连接并传输数据；

(三)实验内容 3：大文件传送

①修改 tcp_apps.c(以及 tcp_stack_trans.py)，使之能够收发文件
②在如图 1 所示拓扑的节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送参照 tcp_stack_trans.py，修改 tcp_apps.c，使之能够收发短消息；
③在一端用 tcp_stack_trans.py 替换 tcp_stack 执行，测试另一端，验证是否能和 linux 协议栈正常建立连接并传输数据；

测试代码如下，需根据不同需求选取自己的 tcp 和 linux 的 tcp：

```
    net.start()
    # h1.cmd('wireshark &')
    # h2.cmd('wireshark &')
    # sleep(3)
    h1.cmd('./tcp_stack server 10001 > h1-output.txt 2>&1 &')
    # h1.cmd('python tcp_stack_trans.py server 10001 > h1-trans.txt 2>&1 &')
    # h1.cmd('python tcp_stack_conn.py server 10001 > h1-conn.txt 2>&1 &')

    h2.cmd('./tcp_stack client 10.0.0.1 10001 > h2-output.txt 2>&1 &')
    # h2.cmd(
    #     'python tcp_stack_trans.py client 10.0.0.1 10001 > h2-trans.txt
2>&1 &')
    # h2.cmd(
    #     'python tcp_stack_conn.py client 10.0.0.1 10001 > h2-conn.txt 2>&1
&')

    sleep(15)

    for r in (h1, h2):
        r.cmd('pkill -SIGTERM tcp_stack')

    # CLI(net)
    net.stop()
```

3. 实验设计

(一)实验内容 1：连接管理

TCP (Transmission Control Protocol)是主机对主机层的传输控制协议，提供可靠的连接服务，TCP 协议为 SOCKET 设计了一些状态，在建立连接和释放连接的时候，SOCKET 会在 状态之间切换。

1、本次实验需要实现三次握手、四次挥手和状态之间的迁移函数状态机

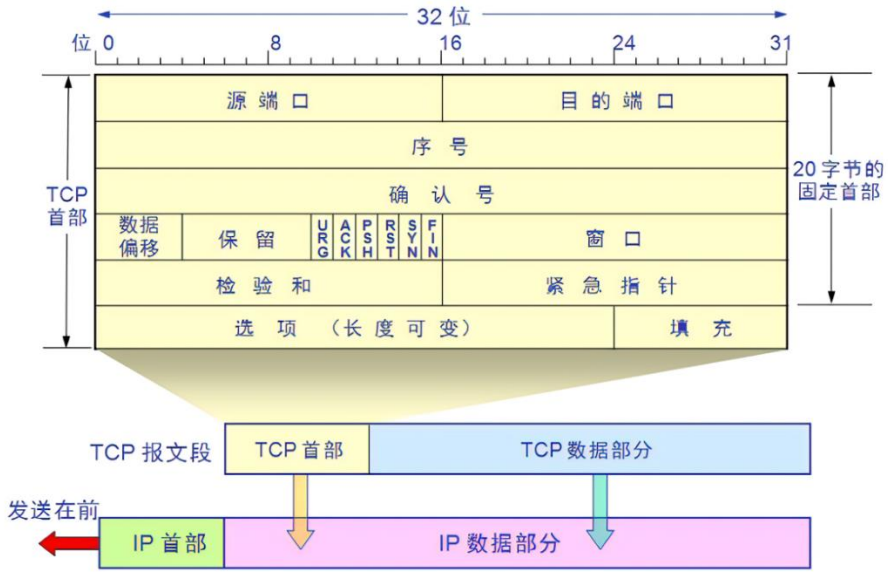


图 2

位码即 tcp 标志位，有 6 种标示：SYN(synchronous 建立联机) ACK(acknowledgement 确认) PSH(push 传送) FIN(finish 结束) RST(reset 重置) URG(urgent 紧急)Sequence number(顺序号码) Acknowledge number(确认号码)

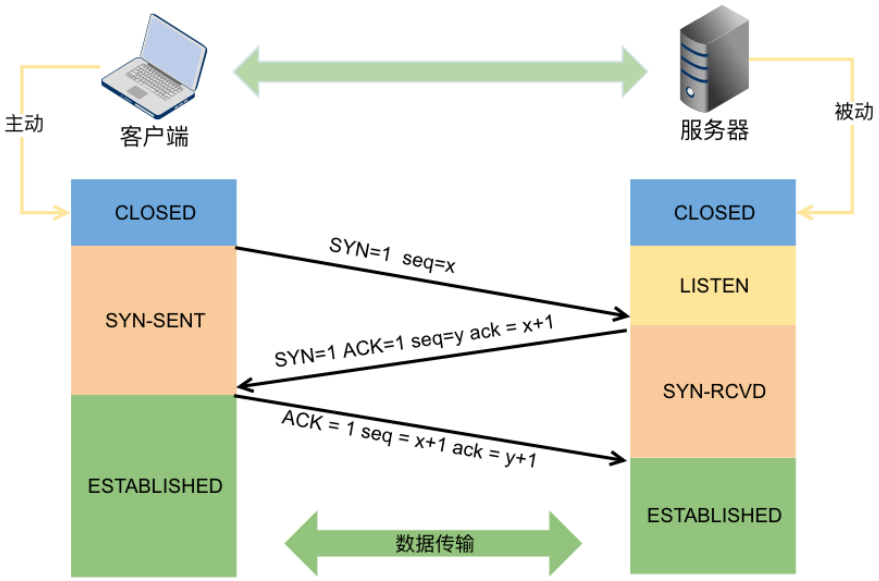


图 3

TCP 三次握手其实就是建立一个 TCP 连接，客户端与服务器交互需要 3 个数据包。握手的主要作用就是为了确认双方的接收和发送能力是否正常，初始序列号，交换窗口大小以及 MSS 等信息：

第一次握手：客户端发送 SYN 报文，并进入 SYN_SENT 状态，等待服务器的确认；

第二次握手：处于 LISTEN 的服务器收到 SYN 报文，需要给客户端发送 ACK 确认报文，同时服务器也要向客户端发送一个 SYN 报文，所以也就是向客户端发送 SYN + ACK 报文，此时服务器进入 SYN_RCVD 状态；

第三次握手：客户端收到 SYN + ACK 报文，向服务器发送确认包，客户端进入 ESTABLISHED 状态。待服务器收到客户端发送的 ACK 包也会进入 ESTABLISHED 状态，完成三次握手。

三次握手过程中，第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据，疯狂着重复发 SYN 报文，这会让服务器花费大量的内存空间来缓存这些报文，这样服务器就更容易被攻击了。对于第三次握手，此时客户端已经处于连接状态，他已经知道服务器的接收、发送能力是正常的了，所以可以携带数据是情理之中。

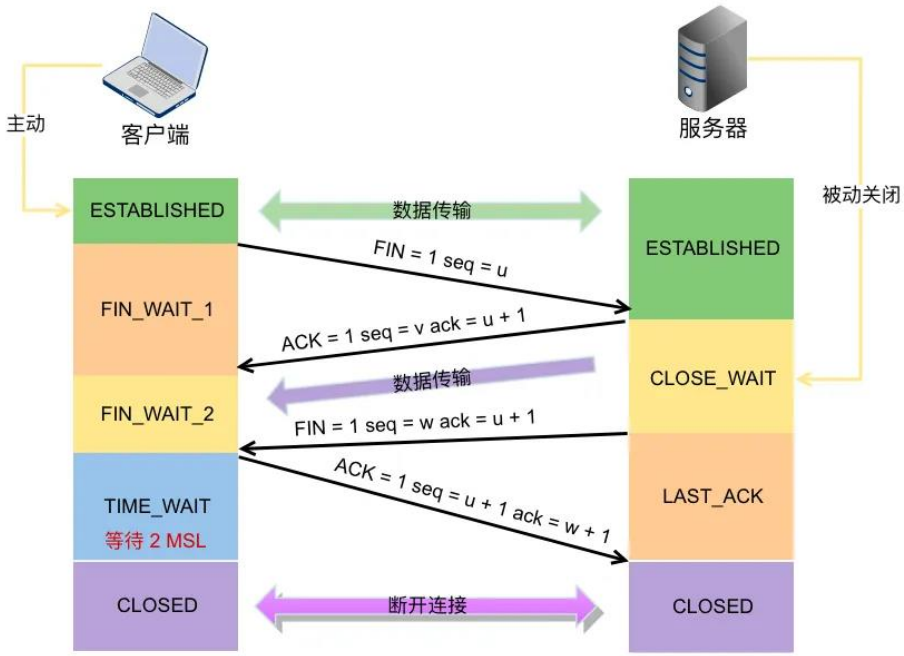


图 4

当我们的应用程序不需要数据通信了，就会发起断开 TCP 连接。建立一个连接需要三次握手，而终止一个连接需要经过四次挥手：

第一次挥手。客户端发起 FIN 包（FIN = 1）,客户端进入 FIN_WAIT_1 状态。TCP 规定，即使 FIN 包不携带数据，也要消耗一个序号。

第二次挥手。服务器端收到 FIN 包，发出确认包 ACK（ack = u + 1），并带上自己的序号 seq=v，服务器端进入了 CLOSE_WAIT 状态。这个时候客户端已经没有数据要发送了，

不过服务器端有数据发送的话，客户端依然需要接收。客户端接收到服务器端发送的 ACK 后，进入了 FIN_WAIT_2 状态。

第三次挥手。服务器端数据发送完毕后，向客户端发送 FIN 包（seq=w ack=u+1），半连接状态下服务器可能又发送了一些数据，假设发送 seq 为 w。服务器此时进入了 LAST_ACK 状态。

第四次挥手。客户端收到服务器的 FIN 包后，发出确认包（ACK=1，ack=w+1），此时客户端就进入了 TIME_WAIT 状态。注意此时 TCP 连接还没有释放，必须经过 2*MSL 后，才进入 CLOSED 状态。而服务器端收到客户端的确认包 ACK 后就进入了 CLOSED 状态，可以看出服务器端结束 TCP 连接的时间要比客户端早一些。

为什么建立连接握手三次，关闭连接时需要是四次呢？其实在 TCP 握手的时候，接收端发送 SYN+ACK 的包是将一个 ACK 和一个 SYN 合并到一个包中，所以减少了一次包的发送，三次完成握手。对于四次挥手，因为 TCP 是全双工通信，在主动关闭方发送 FIN 包后，接收端可能还要发送数据，不能立即关闭服务器端到客户端的数据通道，所以也就不能将服务器端的 FIN 包与对客户端的 ACK 包合并发送，只能先确认 ACK，然后服务器待无需发送数据时再发送 FIN 包，所以四次挥手时必须是四次数据包的交互。

需要注意的是被动连接的一方需要先申请一个 Parent Socket，它只是监听请求的 socket，并不对应一个具体的连接，而它 accept 连接时申请生成的 Child Socket 才是真正对应连接的 Socket，因此一个 Parent Socket 可以对应多个 Child Socket。

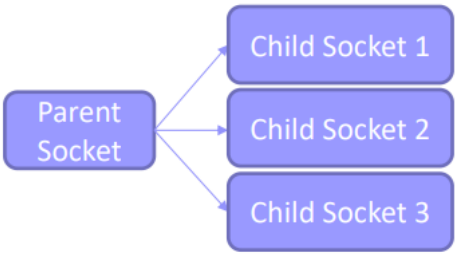


图 5

将其转换为状态机对应代码为：

```
switch (tsk->state)
{
case TCP_LISTEN:
{
    if (cb->flags & TCP_SYN)
    {
        struct tcp_sock *csk = alloc_tcp_sock();
        memcpy((char *)csk, (char *)tsk, sizeof(struct tcp_sock));
        csk->parent = tsk;
        csk->sk_sip = cb->daddr;
        csk->sk_sport = cb->dport;
        csk->sk_dip = cb->saddr;
```

```
            csk->sk_dport = cb->sport;
            csk->iss = tcp_new_iss();
            csk->snd_nxt = csk->iss;
            csk->rcv_nxt = cb->seq + 1;
            tcp_sock_listen_enqueue(csk);
            tcp_set_state(csk, TCP_SYN_RECV);
            tcp_hash(csk); // hash to established_table
            tcp_send_control_packet(csk, TCP_SYN | TCP_ACK);
        }
        return;
    }
    case TCP_SYN_SENT:
    {
        if (cb->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 (cb->flags & TCP_ACK)
        {
            if (tcp_sock_accept_queue_full(tsk))
                return;
            struct tcp_sock *csk = tcp_sock_listen_dequeue(tsk->parent);
            tcp_sock_accept_enqueue(tsk);
            tcp_set_state(tsk, TCP_ESTABLISHED);
            tsk->rcv_nxt = cb->seq;
            tsk->snd_una = cb->ack;
            wake_up(tsk->parent->wait_accept);
        }
        return;
    }
    default:
        break;
    }
```

有 seq 的需要检查 seq 是否有效

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

    switch (tsk->state)
    {
    case TCP_ESTABLISHED:
    {
        if (cb->flags & TCP_FIN)
        {
            tcp_set_state(tsk, TCP_CLOSE_WAIT);
            tsk->rcv_nxt = cb->seq + 1;
            tsk->snd_una = cb->ack;
            tcp_send_control_packet(tsk, TCP_ACK);
            wait_exit(tsk->wait_recv);
            wait_exit(tsk->wait_send);
        }
        else if (cb->flags & TCP_ACK)
        {
            if (cb->pl_len == 0)
            {
                tsk->rcv_nxt = cb->seq;
                tsk->snd_una = cb->ack;
                tcp_update_window_safe(tsk, cb);
            }
            else
                handle_recv_data(tsk, cb);
        }
        break;
    }
    case TCP_LAST_ACK:
    {
        if (cb->flags & TCP_ACK)
        {
            tcp_set_state(tsk, TCP_CLOSED);
            tsk->rcv_nxt = cb->seq;
            tsk->snd_una = cb->ack;
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
        }
        break;
    }
    case TCP_FIN_WAIT_1:
```

```
    {
        if (cb->flags & TCP_ACK)
        {
            tcp_set_state(tsk, TCP_FIN_WAIT_2);
            tsk->rcv_nxt = cb->seq;
            tsk->snd_una = cb->ack;
        }
        break;
    }
    case TCP_FIN_WAIT_2:
    {
        if (cb->flags & TCP_FIN)
        {
            tcp_set_state(tsk, TCP_TIME_WAIT);
            tsk->rcv_nxt = cb->seq + 1;
            tsk->snd_una = cb->ack;
            tcp_send_control_packet(tsk, TCP_ACK);
            tcp_set_timewait_timer(tsk);
        }
        break;
    }
    default:
        break;
    }
```

需要注意初始序列号 ISS 如果是固定的，网络的攻击者很容易猜出后续的确认序号，为了安全起见，避免被第三方猜到从而发送伪造的 RST 报文，因此 ISN 是动态生成的。

2、服务器监听和接受

① 绑定源地址和源端口，哈希到 bind_table。根据参数，设置 tsk 的 backlog 值，转换状态到 LISTEN，并将 tsk 存入 listen table。

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

② 若接收队列空，则等待接收；若接收队列不空，则取出队列的第一个 sock，并接收它。将四元组哈希到 established_table

```
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)
    {
        return child;
    }
    return NULL;
}
```

3.客户端连接：

connect：设置源 IP 地址和源端口，哈希到 bind_table。发送 SYN 请求，进入 SYN_SENT 阶段，sleep 等待应答。在 SYN_SENT 收到 ACK|SYN 时进入 ESTABLISHED 阶段，唤醒 connect 线程。

```
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;
}
```

4.Sock 操作

①对于传入的 TCP sock, 自减其引用统计域 ref_cnt；如果该域减后小于 0，则释放该 sock。

```
void free_tcp_sock(struct tcp_sock *tsk)
{
    if (--tsk->ref_cnt < 0)
    {
        free_wait_struct(tsk->wait_accept);
        free_wait_struct(tsk->wait_connect);
        free_wait_struct(tsk->wait_send);
        free_wait_struct(tsk->wait_recv);
        free_ring_buffer(tsk->rcv_buf);
        free(tsk);
    }
}
```

②首先利用四元组 (saddr, daddr, sport, dport) 查找 established table 中匹配的 sock，若没有则在 listen table 中利用 spoort 查找匹配的 sock:

③关闭 sock . 首先释放占用的资源，然后发送 FIN 包，并且将状态转换到 close。

5.TCP 定时器：

①为每个 sock 设置 timewait timer, 并将其加入 timer_list;

②扫描定时器列表, 对每一个定时器进行老化, 对 TIMEOUT 的定时器: 删除定时器；关闭相应 sock 的 TCP 连接; 将其从 bind_table 中释放，将 sock 从其所属的 table (established table / listen table) 中释放.

```
void tcp_scan_timer_list()
{
    struct tcp_timer *entry = NULL, *q = NULL;
    list_for_each_entry_safe(entry, q, &timer_list, list)
    {
        if (entry->enable == 1 && entry->type == 0 && ((time(NULL) -
entry->timeout) > TCP_TIMEWAIT_TIMEOUT / TCP_MSL))
        {
            struct tcp_sock *tsk = timewait_to_tcp_sock(entry);
            list_delete_entry(&entry->list);
            tcp_set_state(tsk, TCP_CLOSED);
            tcp_unhash(tsk);
            tcp_bind_unhash(tsk);
        }
    }
}
```

这里为什么 TIME_WAIT 状态需要经过 2MSL 才能返回到 CLOSE 状态？首先，MSL 指的是报文在网络中最大生存时间。在客户端发送对服务器端的 FIN 的确认包 ACK 后，这个 ACK 包是有可能不可达的，服务器端如果收不到 ACK 的话需要重新发送 FIN 包。所以客户端发送 ACK 后需要留出 2MSL 时间（ACK 到达服务器 + 服务器发送 FIN 重传包，一来一回）等待确认服务器端确实收到了 ACK 包。也就是说客户端如果等待 2MSL 时间也没有收到服务器端的重传包 FIN，说明可以确认服务器已经收到客户端发送的 ACK。其次可以避免新旧连接混淆。在客户端发送完最后一个 ACK 报文段后，在经过 2MSL 时间，就可以使本连接持续的时间内所产生的所有报文都从网络中消失，使下一个新的连接中不会出现这种旧的连接请求报文。有些自作主张的路由器会缓存 IP 数据包，如果连接重用了，那么这些延迟收到的包就有可能会跟新连接混在一起。

(二)实验内容 2：数据传输

① 调整状态机使得在 established 状态下能正常的处理接收到的数据，实现流量控制：通过调整 recv_window 来表达自己的接收能力。每收到一个数据包，调整当前 sock 的发送窗口为对端的接收窗口大小。调整接收窗口为环形缓存的剩余大小.

```
void handle_recv_data(struct tcp_sock *tsk, struct tcp_cb *cb)
{
    while (ring_buffer_free(tsk->rcv_buf) < cb->pl_len)
    {
        sleep_on(tsk->wait_recv);
    }

    pthread_mutex_lock(&tsk->rcv_buf->lock);
    write_ring_buffer(tsk->rcv_buf, cb->payload, cb->pl_len);
    tsk->rcv_wnd -= cb->pl_len;
    pthread_mutex_unlock(&tsk->rcv_buf->lock);
    wake_up(tsk->wait_recv);
    tsk->rcv_nxt = cb->seq + cb->pl_len;
    tsk->snd_una = cb->ack;
    tcp_send_control_packet(tsk, TCP_ACK);
}
```

② 用户进程读取缓存区数据。若环形缓存为空，则睡眠等待。0 表示读到流结尾，对方断开连接，因此需要判断此时是否断开连接，若状态不再为 TCP_ESTABLISHED 则已断开连接，直接返回 0。在读环形缓存之前先申请缓存的锁。读取后返回实际读取字节数 rlen。

```
int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)
{
    while (ring_buffer_empty(tsk->rcv_buf))
```

```
    {
        if (tsk->state ! = TCP_ESTABLISHED)
            return 0;
        sleep_on(tsk->wait_recv);
    }

    pthread_mutex_lock(&tsk->rcv_buf->lock);
    int rlen = read_ring_buffer(tsk->rcv_buf, buf, len);
    tsk->rcv_wnd = ring_buffer_free(tsk->rcv_buf);
    pthread_mutex_unlock(&tsk->rcv_buf->lock);
    wake_up(tsk->wait_recv);
    return rlen;
}
```

③ 用户进程发送数据。每次发送的大小为 min(MSS,data_left,send_window)，因此需要用多个数据包发送， 当发送窗口为 0 时需要停止发送。

```
int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)
{
    int send_len, packet_len;
    int save_len = len;
    int index = 0;
    while (len)
    {
        int headerlen = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
TCP_BASE_HDR_SIZE;
        send_len = min(len, ETH_FRAME_LEN - headerlen); // 每次读取 1 个数
据包大小的数据
        packet_len = send_len + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
TCP_BASE_HDR_SIZE;
        char *packet = (char *)malloc(packet_len);
        memcpy(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE +
TCP_BASE_HDR_SIZE, buf + index, send_len);
        tcp_send_packet(tsk, packet, packet_len);
        if (tsk->snd_wnd == 0)
            sleep_on(tsk->wait_send);
        len -= send_len;
        index += send_len;
    }
    return save_len;
}
```

4. 实验结果

（1）实验内容一：连接管理

H1 作为 server 输出：

```
DEBUG: find the following interfaces:  h1-eth0.
DEBUG: 0.0.0.0:10001 switch state, from CLOSED to LISTEN.
DEBUG: server 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: server 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: server close.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
```

H2 作为 client 输出：

```
DEBUG: find the following interfaces:  h2-eth0.
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: client connect success.
DEBUG: 10.0.0.2:12345 switch state, from ESTABLISHED to FIN_WAIT-1.
DEBUG: client close.
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.
```

在一端用 tcp_stack.py 替换 tcp_stack 执行，测试另一端得到结果相同。通过 h1 和 h2 的输出可知符合上述 TCP3 次握手和 4 次挥手的过程转换，同时能和 linux 内置的协议栈正常建立连接

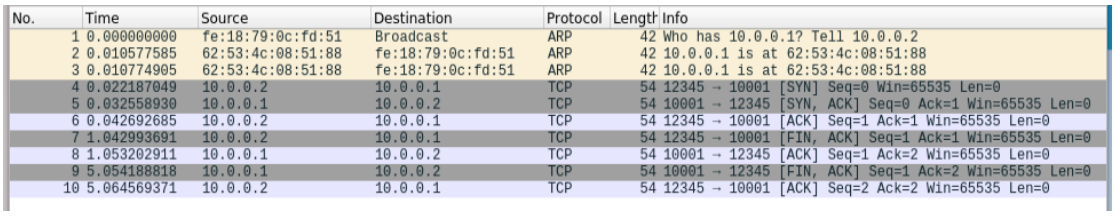
Wireshark 抓包验证：

H1：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	0a:10:c9:83:0a:fa	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010481586	a2:1d:9c:7e:e0:99	0a:10:c9:83:0a:fa	ARP	42	10.0.0.1 is at a2:1d:9c:7e:e0:99
3	0.010507846	a2:1d:9c:7e:e0:99	0a:10:c9:83:0a:fa	ARP	42	10.0.0.1 is at a2:1d:9c:7e:e0:99
4	0.021324105	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.032267473	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.042503501	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.042733594	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.052892525	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.053539101	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.063738840	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

图 6

H2：



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe:18:79:0c:fd:51	Broadcast	ARP	42	Who has 10.0.0.1? Tell 10.0.0.2
2	0.010577585	62:53:4c:08:51:88	fe:18:79:0c:fd:51	ARP	42	10.0.0.1 is at 62:53:4c:08:51:88
3	0.010774905	62:53:4c:08:51:88	fe:18:79:0c:fd:51	ARP	42	10.0.0.1 is at 62:53:4c:08:51:88
4	0.022187049	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
5	0.032558930	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
6	0.042692685	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.042993691	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
8	1.053202911	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=65535 Len=0
9	5.054188818	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
10	5.064569371	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0

图 7

（2）实验二：短消息收发

① client 向 server 发送数据，server 将数据 echo 给 client

Server：

```
DEBUG: find the following interfaces:  h1-eth0.
DEBUG: listen to port 10001.
DEBUG: accept a connection.
DEBUG: Finish transmission.
Client:
DEBUG: find the following interfaces:  h2-eth0.
Routing table of 1 entries has been loaded.
server echoes:
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
server echoes:
123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes:
23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes:
3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes:
456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes:
56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes:
6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes:
789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes:
89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes:
9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
```

② 使用 tcp_stack_trans.py 替换其中任意一端，对端都能正确收发数据

服务器被替换时 h1 作为服务器输出：

```
('10.0.0.2', 12345)
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'str'>
```

客户端被替换时 h2 作为客户端输出：

```
server echoes: 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0
server echoes: 123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01
server echoes: 23456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012
server echoes: 3456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123
server echoes: 456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
server echoes: 56789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345
server echoes: 6789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678
server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

（3）实验三：大文件传送

Md5sum 验证

```
stu@stu:~/computer_net/10-tcp_stack_exp3$ md5sum client-input.dat
bb4e3422bd440e143900252932bdc5d7  client-input.dat
stu@stu:~/computer_net/10-tcp_stack_exp3$ md5sum server-output.dat
bb4e3422bd440e143900252932bdc5d7  server-output.dat
```

Diff 验证

```
stu@stu:~/computer_net/10-tcp_stack_exp3$ diff server-output.dat  client-input.dat
stu@stu:~/computer_net/10-tcp_stack_exp3$
```

文件没有差异，传输成功

5. 实验总结

TCP 是一个十分复杂的协议，本次实验以 TCP 三次握手和四次挥手这个经典问题为主题，初步窥探了 TCP 协议的连接、数据传输和流量控制等知识点。我们知道 TCP 协议是可靠的，而保证数据的可靠需要借助多种方式：

- 分块传送：首先数据被分割成最合适的数据块，
- 等待确认：通过定时器等待接收端发送确认请求，收不到确认则重发
- 确认回复：收到确认后发送确认回复(不是立即发送，通常推迟几分之一秒)
- 数据校验：保持首部和数据的校验和，检测数据传输过程有无变化
- 乱序排序：接收端能重排序数据，以正确的顺序交给应用端
- 重复丢弃：接收端能丢弃重复的数据包
- 流量缓冲：两端有固定大小的缓冲区（滑动窗口），防止速度不匹配丢数据

参考资料：

【1】三次握手。
【2】四次挥手。
【3】Socket 技术详解。
【4】TCP 协议详解。