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

# 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。

# 实验流程

1. **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.

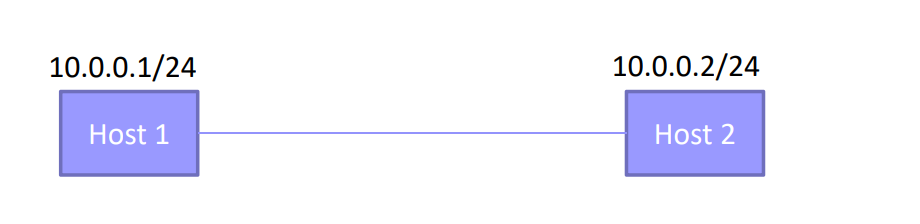


图1

1. **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；

1. **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)*

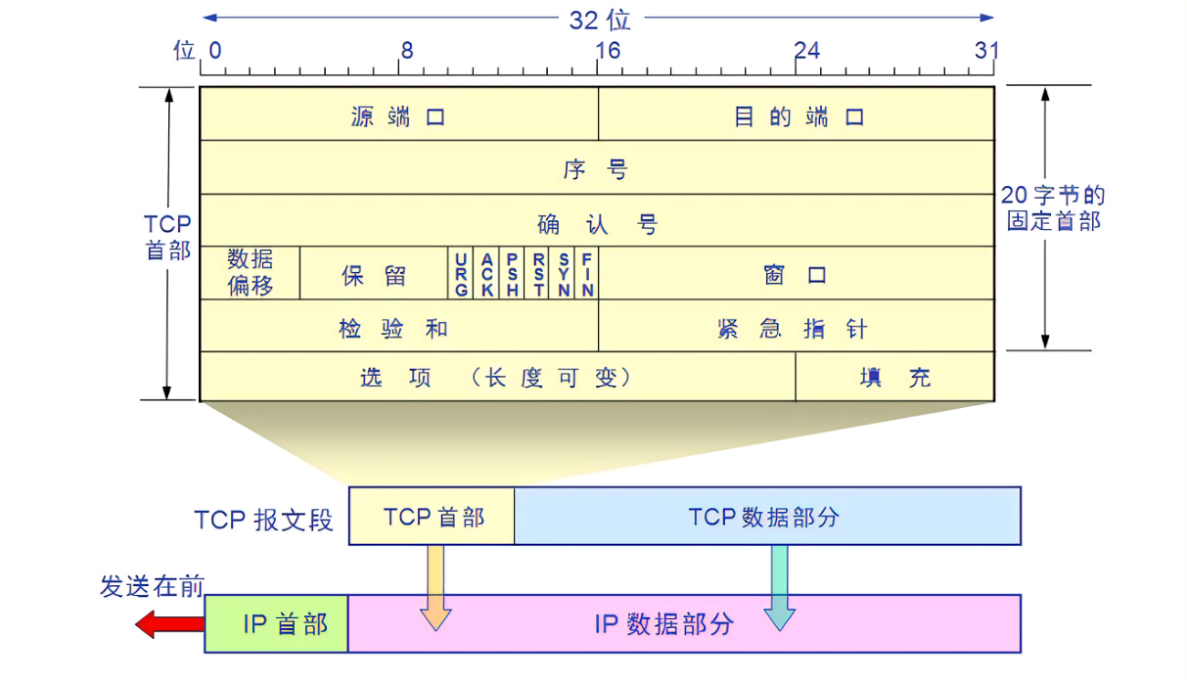
    net.stop()

# 实验设计

1. **实验内容1：连接管理**

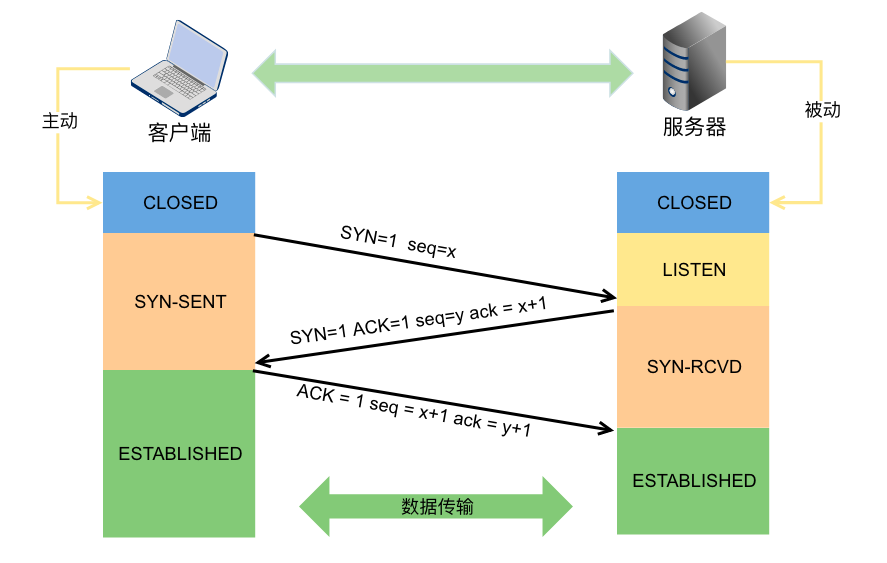
TCP (Transmission Control Protocol)是主机对主机层的传输控制协议，提供可靠的连接服务，TCP协议为SOCKET设计了一些状态，在建立连接和释放连接的时候，SOCKET会在状态之间切换。

**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**

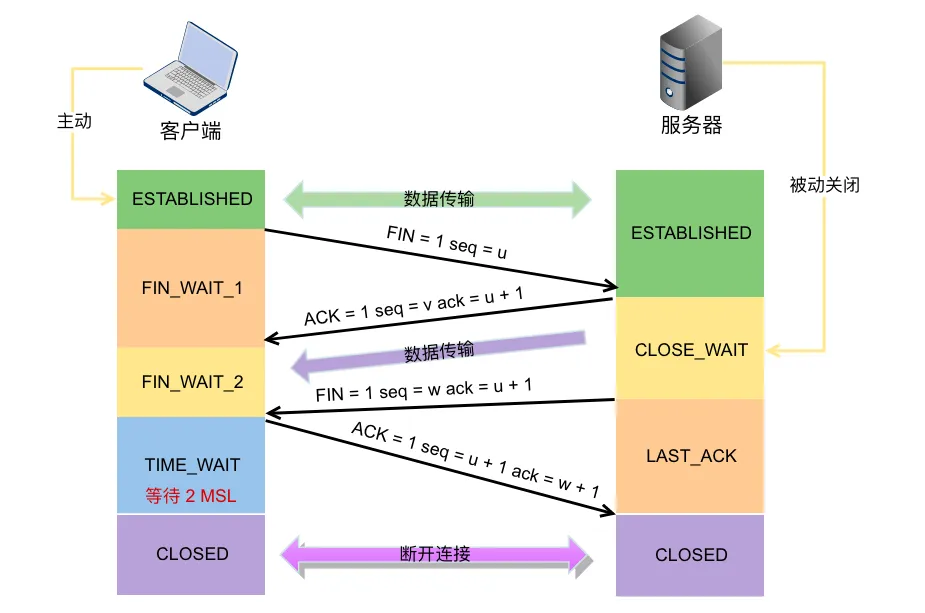
TCP 三次握手其实就是建立一个 TCP 连接，客户端与服务器交互需要 3 个数据包。握手的主要作用就是为了确认双方的接收和发送能力是否正常，初始序列号，交换窗口大小以及 MSS 等信息：

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。

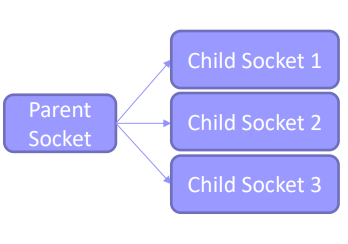
第二次挥手。服务器端收到 FIN 包，发出确认包 ACK（ack = u + 1），并带上自己的序号 seq=v，服务器端进入了 CLOSE\_WAIT 状态。这个时候客户端已经没有数据要发送了，不过服务器端有数据发送的话，客户端依然需要接收。客户端接收到服务器端发送的 ACK 后，进入了 FIN\_WAIT\_2 状态。

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。

1. **实验内容2：数据传输**
2. 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);

}

1. 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;

}

1. 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;

}

# 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：

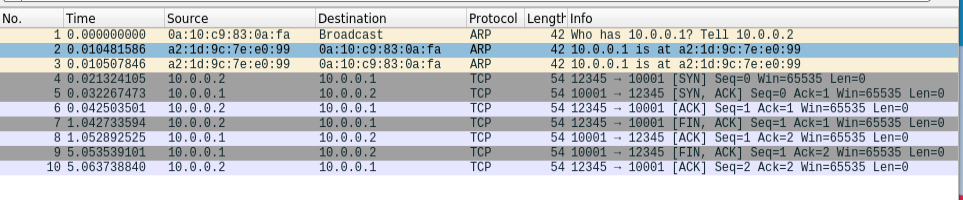


图6

H2：

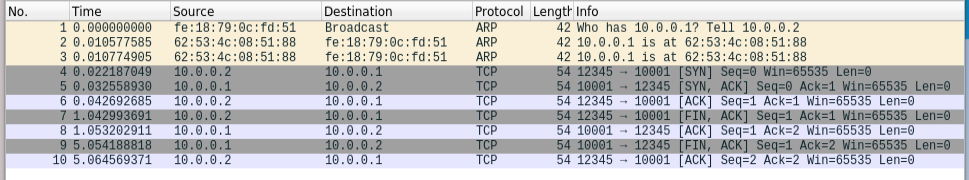


图7

**（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

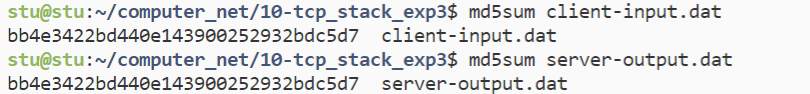
server echoes: 789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567

server echoes: 89abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345678

server echoes: 9abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

**（3）实验三：大文件传送**

Md5sum验证



Diff验证



文件没有差异，传输成功

# 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](https://blog.csdn.net/O9A0MA/article/details/90647803)。

【2】[四次挥手](https://blog.csdn.net/O9A0MA/article/details/90731748)。

【3】[Socket技术详解](https://www.jianshu.com/p/066d99da7cbd)。

【4】[TCP协议详解](https://juejin.cn/post/6844903685563105293)。