张芙菻

2020K8009929012

网络传输机制实验

1. **实验内容**

**1.1实现TCP协议栈，节点之间能够在TCP协议之上正确建立连接和传输数据。**

**1.2完成应用程序的编写，完成节点之间传输短消息和文件的操作。**

1. **实验实现**

在网络传输机制实验中，我们需要实现最基本的TCP连接管理功能，使得节点之间能够在无丢包网络环境中建立和断开连接；实现最基本的TCP数据收发功能，使得节点之间能够在无丢包网络环境中传输数据。

**2.1 TCP协议分析**

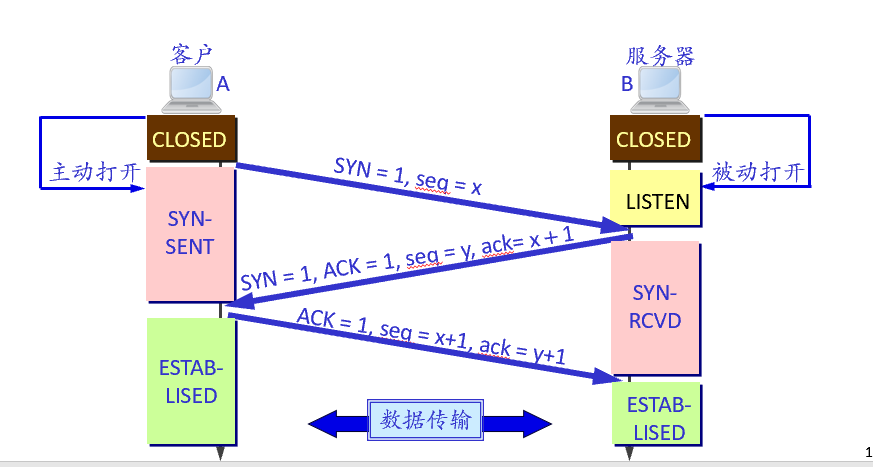
TCP协议提供端到端的、面向连接的、可靠的、有序的字节流传输服务。TCP协议实现点对点的双工通信，TCP将应用程序交付下来的数据看作一连串无结构的有序的字节流，每条TCP连接是一对点对点字节流，每个方向一个字节流，TCP两端设有发送缓存和接收缓存，存放双向通信的数据：发送时，应用进程把数据传入TCP的发送缓存，TCP在合适时候将数据发出；接收时，TCP把数据放入接收缓存，应用进程在合适时候读取数据。

**2.1.1 TCP建立连接**

在TCP通信的过程中，通信双方先建立逻辑连接，再进行双向数据流传输，通信结束后撤销连接，同时利用滑动窗口机制实现可靠传输——对一个连接上传输的每个字节编号，通过接收确认和重传保证可靠传输；流量控制——防止发送方发送的数据超出接收方的接收能力；拥塞控制——防止过多数据注入网络造成网络结点或链路超载。

TCP通过三次握手机制建立连接：

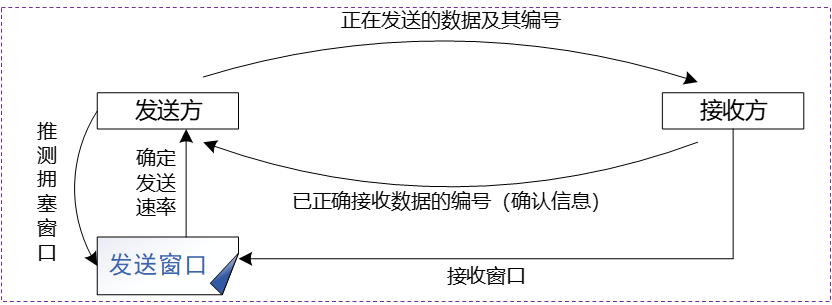
1. 最初，两端的TCP进程都处于CLOSED状态
2. 服务器B进程被动打开连接，进入LISTEN状态，等待客户进程的连接请求
3. TCP客户进程A主动打开连接，创建传输控制块TCB，向B发送连接请求报文段，进入SYN-SENT状态。
4. B收到请求后，应答确认报文段，进入SYN-RCVD状态。
5. A收到B的确认后，向B应答确认，进入ESTABLISHED状态。
6. B收到A的确认后，也进入ESTABLISHED状态
7. 连接建立



**2.1.2 TCP数据传输**

在TCP协议中，双方可以同时发送数据，有特定序列号值的数据，从发送方向接收方流动，发送数据的过程中每字节顺序编号，每个报文段中的序列号值指的是本报文段所发送的数据的第一个字节的序号。

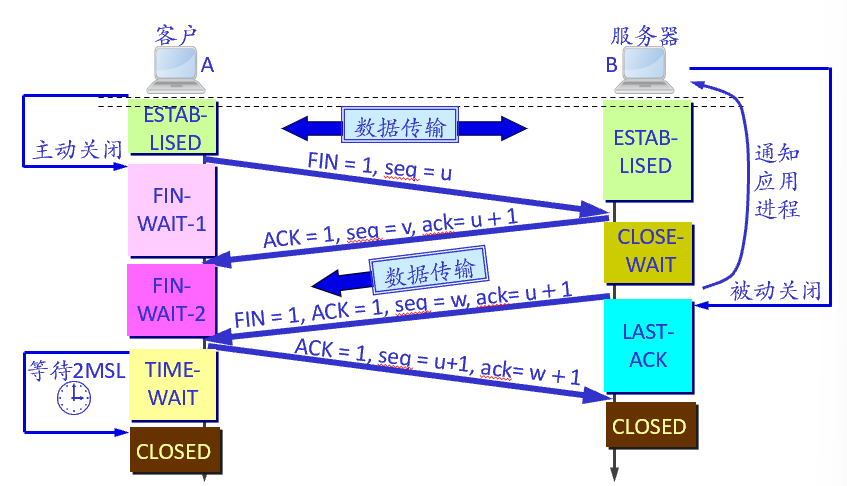
对数据的接收确认、接收窗口大小由接收方向发送方应答。



**2.1.3 TCP释放连接**

TCP连接的任何一方都可以通过发送FIN报文段主动关闭连接：

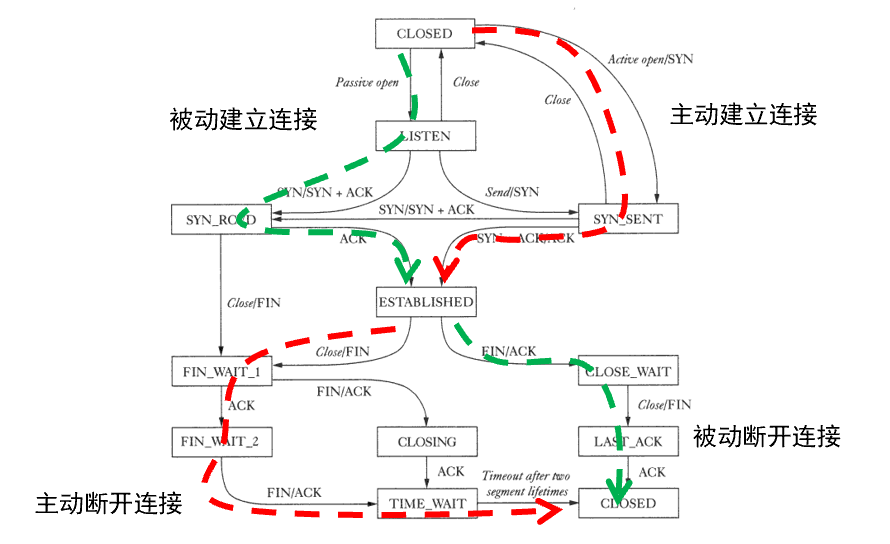
1. A、B都处于ESTABLISHED状态，A向B发送释放报文段，进入FIN-WAIT-1状态
2. B收到A的连接释放报文段后，应答确认，进入CLOSE-WAIT状态
3. A收到B的确认后，进入FIN-WAIT-2状态，B到A方向的连接未关闭，B若发送数据，A仍要接收
4. 若B已没有向A发送的数据，其应用进程通知TCP释放连接
5. B向A发送连接释放报文段后，进入LAST-ACK状态
6. A收到B的连接释放报文段后，回复确认，进入TIME-WAIT状态
7. B收到A的确认后，撤销传输控制块TCB，（等待超过2MSL后）进入CLOSED状态



**2.2 tcp协议栈实现**

**2.2.1状态机实现**

根据之前对tcp协议建立连接以及释放连接过程的分析，我们得到TCP状态机的状态转换图：



在本实验中我们实现虚线标注的部分，具体代码实现如下所示：

void tcp\_process(struct tcp\_sock \*tsk, struct tcp\_cb \*cb, char \*packet)

{

    if (cb->flags & TCP\_RST) {

        tcp\_sock\_close(tsk);

        return;

    }

    if (cb->flags & TCP\_ACK) {

        tsk->snd\_una = cb->ack;

    }

    tsk->rcv\_nxt = cb->seq\_end;

    switch (tsk->state) {

    case TCP\_LISTEN:

        if (cb->flags == TCP\_SYN) {

            struct tcp\_sock \*csk = alloc\_child\_tcp\_sock(tsk, cb);

            tcp\_send\_control\_packet(csk, TCP\_SYN | TCP\_ACK);

            tcp\_set\_state(csk, TCP\_SYN\_RECV);

        }

        break;

    case TCP\_SYN\_SENT:

        if (cb->flags == (TCP\_SYN | TCP\_ACK)) {

            tcp\_update\_window\_safe(tsk, cb);

            tcp\_send\_control\_packet(tsk, TCP\_ACK);

            tcp\_set\_state(tsk, TCP\_ESTABLISHED);

            wake\_up(tsk->wait\_connect);

        }

        break;

    case TCP\_SYN\_RECV:

        if (cb->flags == TCP\_ACK) {

            tcp\_sock\_accept\_enqueue(tsk);

            tcp\_set\_state(tsk, TCP\_ESTABLISHED);

            wake\_up(tsk->wait\_accept);

        }

        break;

    case TCP\_ESTABLISHED:

        if (cb->flags & TCP\_FIN) {

            tcp\_send\_control\_packet(tsk, TCP\_ACK);

            tcp\_set\_state(tsk, TCP\_CLOSE\_WAIT);

            if (tsk->wait\_empty->sleep) {

                wake\_up(tsk->wait\_empty);

            }

        }

        if (cb->flags & TCP\_ACK) {

            tcp\_update\_window\_safe(tsk, cb);

            if (cb->pl\_len != 0) {

                pthread\_mutex\_lock(&tsk->rcv\_buf\_lock);

                while (ring\_buffer\_free(tsk->rcv\_buf) < cb->pl\_len) {

                    pthread\_mutex\_unlock(&tsk->rcv\_buf\_lock);

                    sleep\_on(tsk->wait\_full);

                    pthread\_mutex\_lock(&tsk->rcv\_buf\_lock);

                }

                write\_ring\_buffer(tsk->rcv\_buf, cb->payload, cb->pl\_len);

                tsk->rcv\_wnd = ring\_buffer\_free(tsk->rcv\_buf);

                pthread\_mutex\_unlock(&tsk->rcv\_buf\_lock);

                if (tsk->wait\_empty->sleep) {

                    wake\_up(tsk->wait\_empty);

                }

                tcp\_send\_control\_packet(tsk, TCP\_ACK);

            }

        }

        break;

    case TCP\_FIN\_WAIT\_1:

        if (cb->flags == TCP\_ACK) {

            tcp\_set\_state(tsk, TCP\_FIN\_WAIT\_2);

        } else if (cb->flags == (TCP\_ACK | TCP\_FIN)) {

            tcp\_send\_control\_packet(tsk, TCP\_ACK);

            tcp\_set\_state(tsk, TCP\_TIME\_WAIT);

            tcp\_set\_timewait\_timer(tsk);

        }

        break;

    case TCP\_FIN\_WAIT\_2:

        if (cb->flags & TCP\_FIN) {

            tcp\_send\_control\_packet(tsk, TCP\_ACK);

            tcp\_set\_state(tsk, TCP\_TIME\_WAIT);

            tcp\_set\_timewait\_timer(tsk);

        }

        break;

    case TCP\_LAST\_ACK:

        if (cb->flags & TCP\_ACK) {

            tcp\_unhash(tsk);

            tcp\_set\_state(tsk, TCP\_CLOSED);

        }

        break;

    default:

        break;

    }

}

**2.2.2 tcp\_sock管理函数实现**

free\_tcp\_sock函数释放tcp\_sock所占用的所有资源，为了确保协议栈的安全运行，每次socket被使用时，ref\_cnt+1，每次调用free\_tcp\_sock函数时，ref\_cnt-1，当ref\_cnt减至0时，释放所有的资源。

void free\_tcp\_sock(struct tcp\_sock \*tsk)

{

    tsk->ref\_cnt -= 1;

    if (tsk->ref\_cnt <= 0) {

        free\_wait\_struct(tsk->wait\_accept);

        free\_wait\_struct(tsk->wait\_connect);

        free\_wait\_struct(tsk->wait\_empty);

        free\_wait\_struct(tsk->wait\_full);

        free\_wait\_struct(tsk->wait\_send);

        free\_ring\_buffer(tsk->rcv\_buf);

        free(tsk);

    }

}

tcp\_sock\_lookup\_established函数利用四元组信息（saddr，daddr，sport，dport）在established\_table中查找tcp\_sock，如果找到了，则返回相应的socket，如果没有，则返回NULL。

具体代码实现如下所示：

struct tcp\_sock \*tcp\_sock\_lookup\_established(u32 saddr, u32 daddr, u16 sport, u16 dport)

{

    int idx = tcp\_hash\_function(saddr, daddr, sport, dport);

    struct tcp\_sock \*ptr;

    list\_for\_each\_entry (ptr, &tcp\_established\_sock\_table[idx], hash\_list) {

        if (saddr == ptr->sk\_sip && sport == ptr->sk\_sport &&

            daddr == ptr->sk\_dip && dport == ptr->sk\_dport) {

            return ptr;

        }

    }

    return NULL;

}

tcp\_sock\_lookup\_listen函数的实现与tcp\_sock\_lookup\_established函数类似，同样利用四元组信息在listen\_table中查找socket，具体实现代码如下所示：

struct tcp\_sock \*tcp\_sock\_lookup\_listen(u32 saddr, u16 sport)

{

    int idx = tcp\_hash\_function(0, 0, sport, 0);

    struct tcp\_sock \*ptr;

    list\_for\_each\_entry (ptr, &tcp\_listen\_sock\_table[idx], hash\_list) {

        if (sport == ptr->sk\_sport) {

            return ptr;

        }

    }

    return NULL;

}

tcp\_sock\_connect函数主动发起连接，其首先确定四元组信息，分配源端口，然后发送SYN数据包，转到TCP\_SYN\_SENT状态，将socket加入到哈希表中，唤醒wait\_connect进程，等待客户端响应。

具体代码实现如下所示：

int tcp\_sock\_connect(struct tcp\_sock \*tsk, struct sock\_addr \*skaddr)

{

    tsk->sk\_dip = ntohl(skaddr->ip);

    tsk->sk\_dport = ntohs(skaddr->port);

    tsk->sk\_sip = longest\_prefix\_match(ntohl(skaddr->ip))->iface->ip;

    if (tcp\_sock\_set\_sport(tsk, 0) < 0) {

        log(ERROR, "No available port");

        return -1;

    }

    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 tsk->sk\_sport;

}

tcp\_sock\_accept函数首先判断接收队列是否为空，如果为空，唤醒wait\_connect进程，等待接收接下来的连接请求，否则弹出第一个tcp\_sock并且接收它，这里调用tcp\_sock\_accept\_dequeue函数完成该操作，具体实现代码如下所示：

struct tcp\_sock \*tcp\_sock\_accept(struct tcp\_sock \*tsk)

{

    while (list\_empty(&tsk->accept\_queue)) {

        sleep\_on(tsk->wait\_accept);

    }

    return tcp\_sock\_accept\_dequeue(tsk);

}

tcp\_sock\_close函数完成主动断开连接处理和被动断开连接处理，当此时为TCP\_CLOSE\_WAIT状态时，为被动断开连接，发送FIN&ACK数据包，将状态设置为TCP\_LAST\_ACK，等待连接关闭；当此时是TCP\_CLOSE\_WAIT状态时，主动断开连接，发送FIN&ACK数据包，将状态设置为TCP\_FIN\_WAIT\_1，等待对方的确认信号，具体代码实现如下所示：

void tcp\_sock\_close(struct tcp\_sock \*tsk)

{

    if (tsk->state == TCP\_CLOSE\_WAIT) {

        tcp\_send\_control\_packet(tsk, TCP\_FIN | TCP\_ACK);

        tcp\_set\_state(tsk, TCP\_LAST\_ACK);

    } else if (tsk->state == TCP\_ESTABLISHED) {

        tcp\_send\_control\_packet(tsk, TCP\_FIN | TCP\_ACK);

        tcp\_set\_state(tsk, TCP\_FIN\_WAIT\_1);

    } else {

        tcp\_unhash(tsk);

        tcp\_set\_state(tsk, TCP\_CLOSED);

    }

}

**2.2.3 TCP协议栈数据收发函数实现**

tcp\_sock\_read函数将环形缓存中的数据读取到应用程序中，返回正在读取的数据长度，具体实现代码如下所示：

int tcp\_sock\_read(struct tcp\_sock \*tsk, char \*buf, int len) {

    pthread\_mutex\_lock(&tsk->rcv\_buf\_lock);

    while (ring\_buffer\_empty(tsk->rcv\_buf)) {

        pthread\_mutex\_unlock(&tsk->rcv\_buf\_lock);

        if (tsk->state == TCP\_CLOSE\_WAIT) {

            return 0;

        }

        sleep\_on(tsk->wait\_empty);

        if (tsk->state == TCP\_CLOSE\_WAIT) {

            if (!ring\_buffer\_empty(tsk->rcv\_buf)) {

                pthread\_mutex\_lock(&tsk->rcv\_buf\_lock);

            } else {

                return 0;

            }

        } else {

            pthread\_mutex\_lock(&tsk->rcv\_buf\_lock);

        }

    }

    int rcv\_len = read\_ring\_buffer(tsk->rcv\_buf, buf, len);

    tsk->rcv\_wnd = ring\_buffer\_free(tsk->rcv\_buf);

    if (ring\_buffer\_free(tsk->rcv\_buf) > 0 && tsk->wait\_full->sleep) {

        wake\_up(tsk->wait\_full);

    }

    pthread\_mutex\_unlock(&tsk->rcv\_buf\_lock);

    return rcv\_len;

}

tcp\_sock\_write函数将应用程序中的数据组成数据包发送出去，返回发送数据的长度，具体实现代码如下所示：

int tcp\_sock\_write(struct tcp\_sock \*tsk, char \*buf, int len) {

    int len\_write = len;

    while (len > 0) {

        int len\_sent = tcp\_send\_data(tsk, buf, len);

        buf += len\_sent;

        len -= len\_sent;

    }

    return len\_write;

}

int tcp\_send\_data(struct tcp\_sock \*tsk, char \*buf, int len) {

    len = min(len, ETH\_FRAME\_LEN - ETHER\_HDR\_SIZE - IP\_BASE\_HDR\_SIZE - TCP\_BASE\_HDR\_SIZE);

    int pkt\_len = ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE + TCP\_BASE\_HDR\_SIZE + len;

    char \*pkt = malloc(pkt\_len);

    memcpy(pkt + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE + TCP\_BASE\_HDR\_SIZE, buf, len);

    while (tsk->snd\_wnd < len) {

        tsk->snd\_wnd = 0;

        sleep\_on(tsk->wait\_send);

    }

    tcp\_send\_packet(tsk, pkt, pkt\_len);

    return len;

}

**2.2.4 TCP定时器实现**

tcp\_scan\_timer\_list函数扫描定时器列表，查找每个存在时间超过2MSL的tcp\_sock并且释放它，具体实现代码如下所示：

void tcp\_scan\_timer\_list()

{

    struct tcp\_timer \*p, \*q;

    list\_for\_each\_entry\_safe (p, q, &timer\_list, list) {

        p->timeout += TCP\_TIMER\_SCAN\_INTERVAL;

        if (p->enable == 1 && p->timeout >= TCP\_TIMEWAIT\_TIMEOUT) {

            list\_delete\_entry(&p->list);

            struct tcp\_sock \*tsk = timewait\_to\_tcp\_sock(p);

            tcp\_set\_state(tsk, TCP\_CLOSED);

            tcp\_bind\_unhash(tsk);

            tcp\_unhash(tsk);

        }

    }

}

tcp\_set\_timewait\_timer函数设置socket定时器，并将其添加到定时器列表中，具体代码实现如下所示：

void tcp\_set\_timewait\_timer(struct tcp\_sock \*tsk)

{

    tsk->timewait.enable = 1;

    tsk->timewait.type = 0;

    tsk->timewait.timeout = 0;

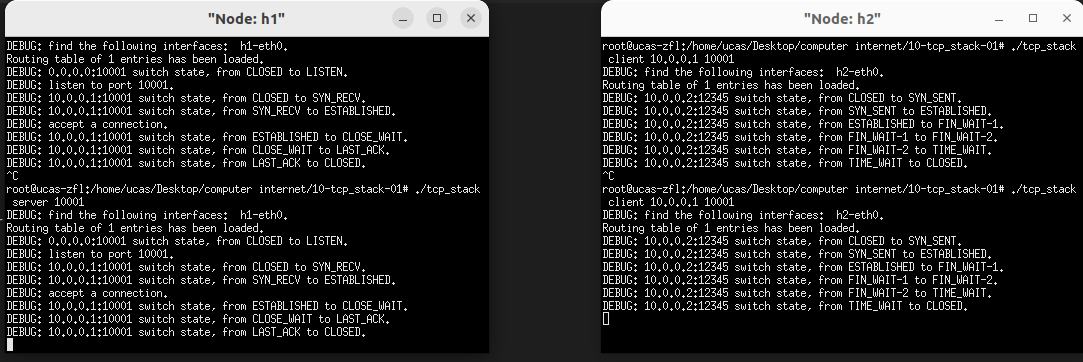
    list\_add\_tail(&tsk->timewait.list, &timer\_list);

}

1. **实验结果**

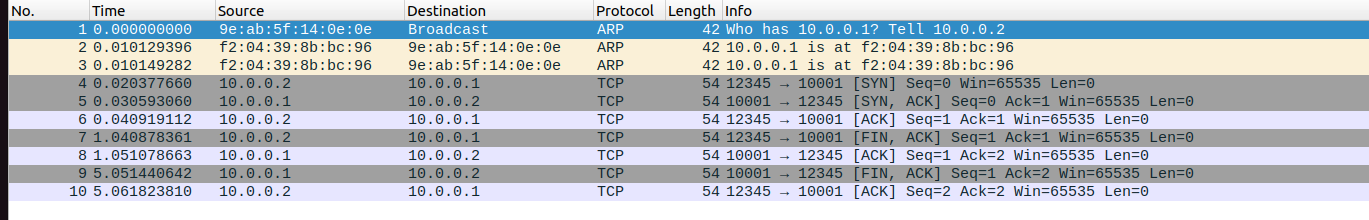
**3.1连接管理**

**3.1.1在节点h1 h2 上执行tcp\_stack**



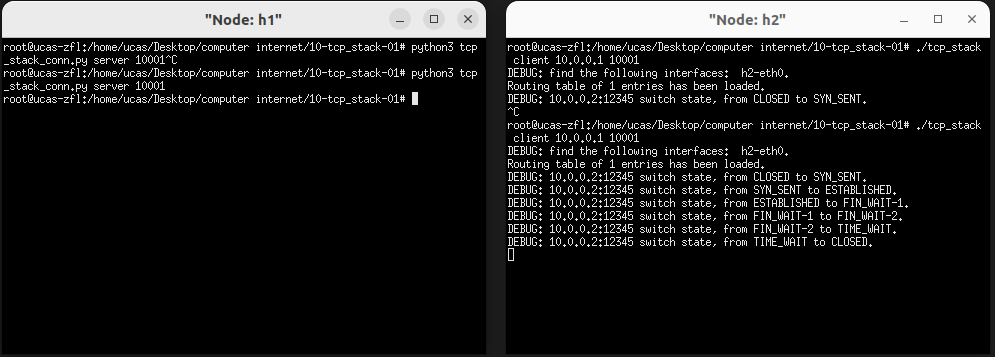
如上图所示，连接成功建立之后成功断开。

wireshark抓包结果：



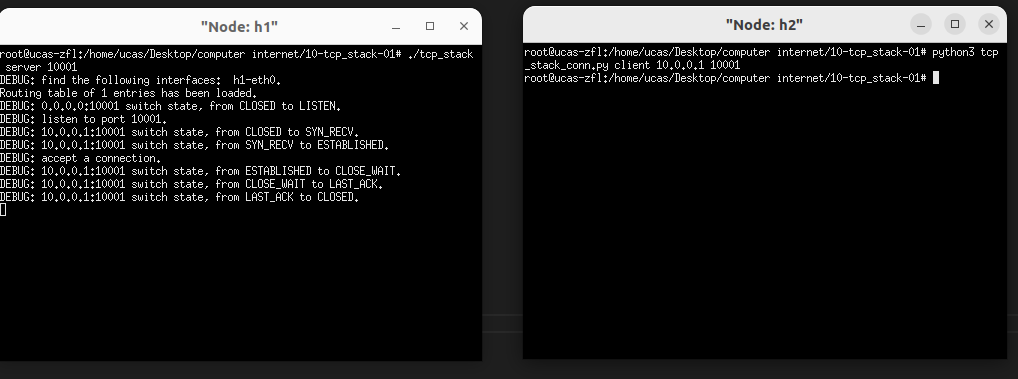
通过wireshark我们看到发送数据包的过程符合我们之前分析的连接建立与释放的过程。

**3.1.2利用tcp\_stack\_conn.py替换服务端**



客户端工作正常。

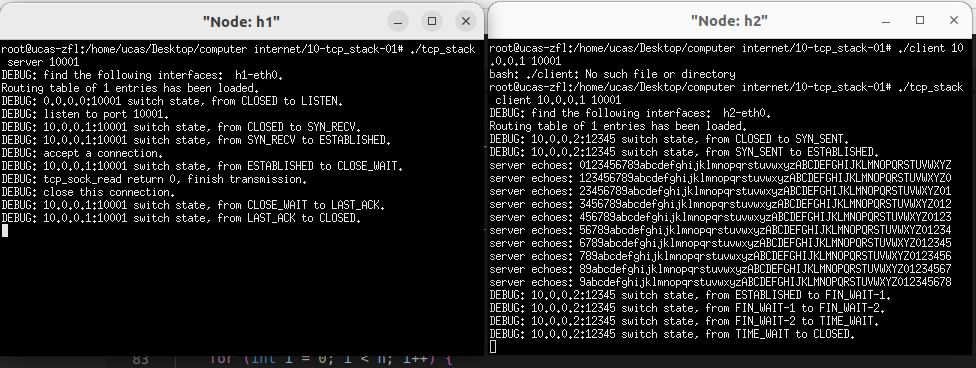
**3.1.3利用tcp\_stack\_conn.py替换客户端**



服务端工作正常。

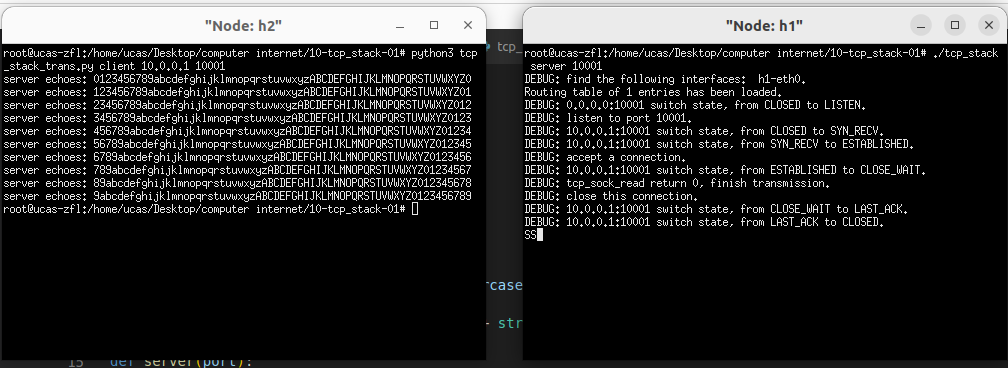
**3.2短消息收发**

**3.2.1在h1 h2节点上执行tcp\_stack**



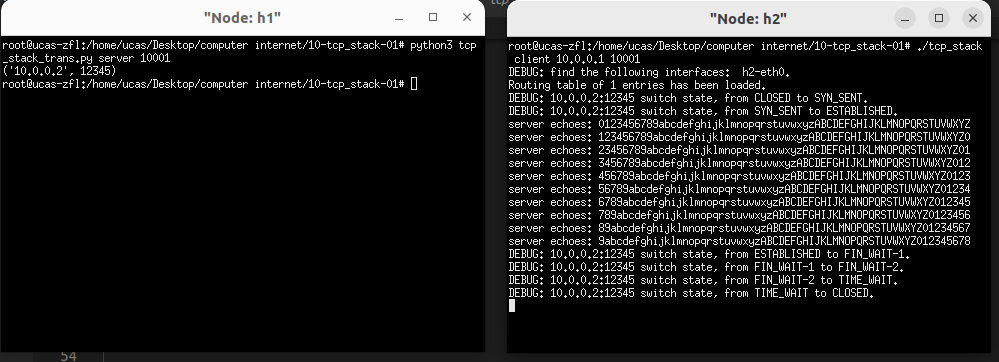
从图中我们看到，客户端和服务端之前成功完成了消息的收发。

**3.2.2利用tcp\_stack\_trans.py替换客户端**



服务端工作正常。

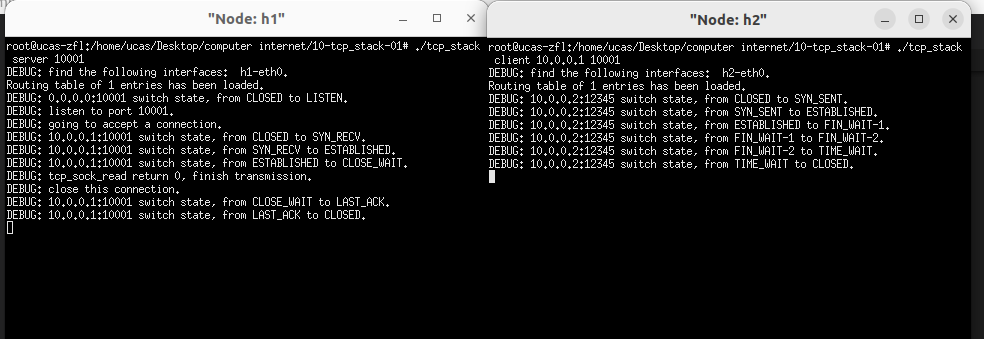
**3.2.3利用tcp\_stack\_trans.py替换服务端**



服务端工作正常。

**3.3大文件传送**

**3.3.1 h1 h2节点上均执行tcp\_stack**

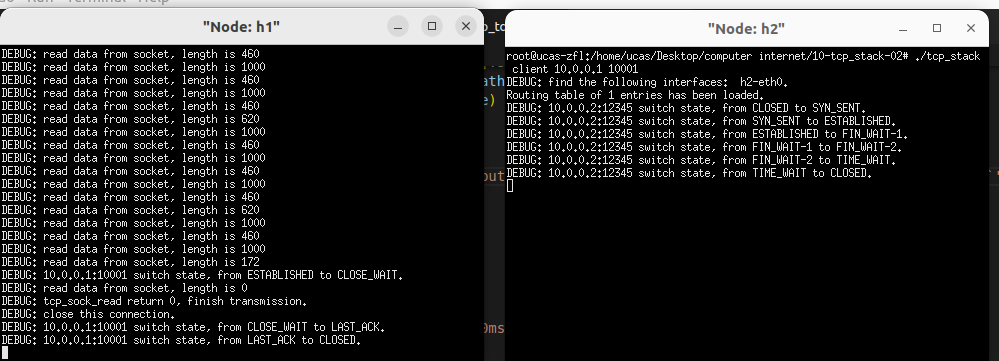


节点之间完成了大文件的传送。



利用md5sum比较两个文件，两个文件完全相同。

**3.3.2利用tcp\_stack\_trans.py替换服务端**



客户端工作正常。



两个文件完全相同。