

网络传输机制实验一报告

李昊宸

2017K8009929044

（一）TCP 连接状态实现

一、实验内容

1. TCP server client 实验：

1) 运行网络拓扑(tcp_topo.py)

2) 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

3) 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关闭连接

2. TCP 交互验证 server 实验：

1) 运行网络拓扑(tcp_topo.py)

2) 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

3) 在节点 h2 上执行标准 client 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 上运行标准 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关闭连接

4) 通过 wireshark 抓包来验证建立和关闭连接的正确性

3. TCP 交互验证 client 实验:

1) 运行网络拓扑(tcp_topo.py)

2) 在节点 h1 上执行标准 server 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 上运行标准 TCP 协议栈的服务器模式

3) 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关闭连接

4) 通过 wireshark 抓包来验证建立和关闭连接的正确性

二、实验流程

1. 搭建实验环境

arp.c arpcache.c icmp.c ip.c main.c packet.c rtable.c
rtable_internal.c
tcp_apps.c : 基于 tcp-stack 的服务器和客户端程序
tcp.c : TCP 协议相关处理函数
tcp_in.c : TCP 接收相关函数
tcp_out.c : TCP 发送相关函数
tcp_sock.c : tcp_sock 操作相关函数
tcp_stack.py : python 应用实现, 用于测试
tcp_timer.c : TCP 定时器
tcp_topo.py : 实现二节点的简单拓扑



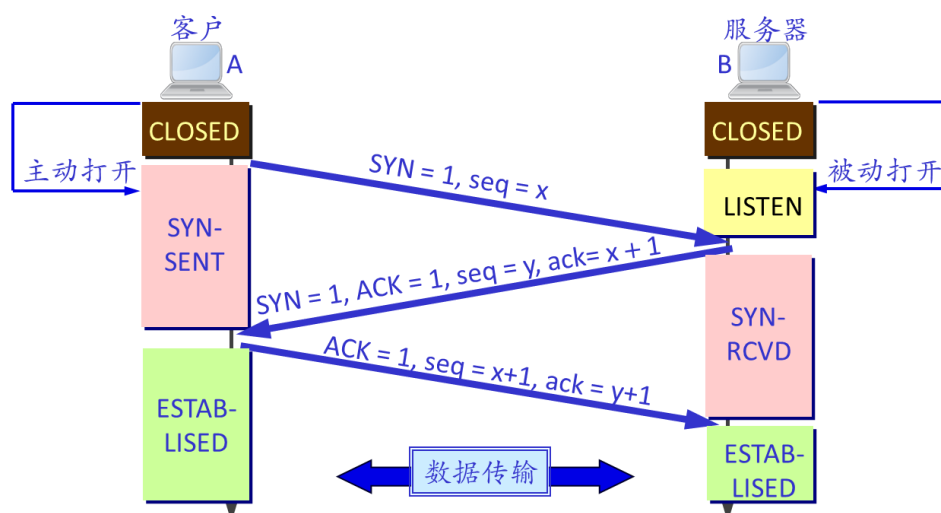
图一 二节点网络拓扑

TCP: Transmission Control Protocol,

传输控制协议，面向连接的、可靠的、基于字节流的传输层通信协议。

本次实验需要实现的是 TCP 建立和释放，也就是要实现 TCP 状态机。

连接建立三次握手：



图二 TCP 连接建立

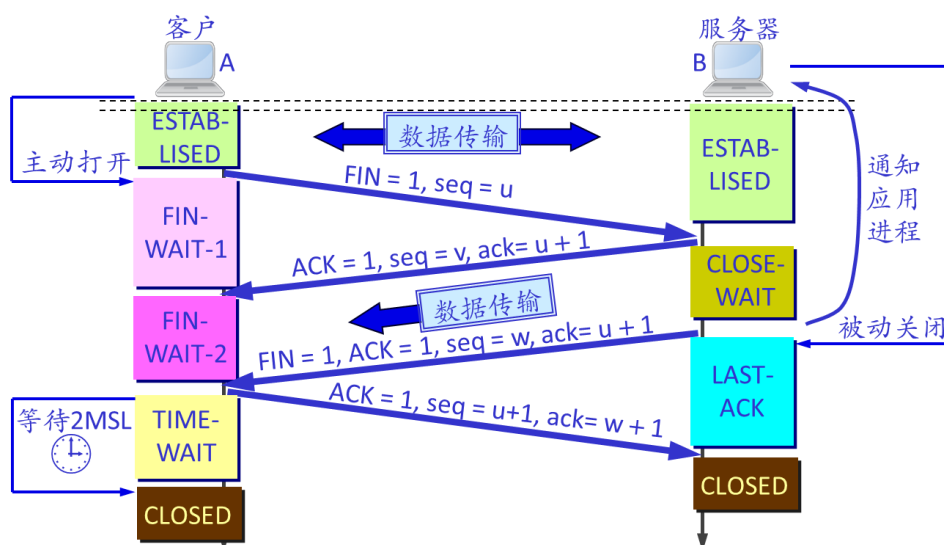
1) TCP 客户进程 A 主动打开连接：创建 TCP_sock，向 B 发送连接请求报文段，请求报文段首部的同步位 SYN 置 1，随机选择初始序号 SequenceNum 为 x。随后进入 SYN-SENT（同步已发送）状态，并陷入睡眠（connect 操作），等待对端报文。

2) B 收到请求后，创建子 sock 做出应答：应答确认报文段，确认报文段首部的标志位 SYN、ACK 都置 1，确认号 Acknowledgment = x+1，并随机选择自己的初始序号为 y。随后子 sock 进入 SYN-RCVD (同步收到)状态，注意此时主 sock 在此步之前已经陷入睡眠（accept 操作），等待对端报文。

3) A 收到 B 的确认后，向 B 应答确认，该确认报文段首部的标志位 ACK 置 1，确认号为 y+1，序号为 x+1。B 收到 A 的确认后，也进入 ESTABLISHED (已建立连接)状态。需要注意的是，该报文若不含数据，就不消耗序列号。随后进入 ESTABLISHED (已建立连接)状态，并从 connect 中被唤醒。

4) B 收到 A 的 ACK 后，子 sock 状态转移到 ESTABLISHED，同时主 sock 从 accept 中被唤醒，将子 sock 出队列，做好相应工作后再次调用 accept，等待下个连接。

连接释放四次握手:



图三 TCP 连接释放

1) TCP 客户进程 A 主动关闭连接: A 向 B 发送连接释放报文段, 进入 FIN-WAIT-1 (终止等待 1) 状态。连接释放报文段首部的终止控制位 FIN 置 1, 序号 u 等于 A 前面已传输过的数据的最后一个字节的序号加 1。需要注意的是, FIN 报文段即使不携带数据, 也要消耗掉一个序号。

2) B 收到 A 的连接释放报文段后, 应答确认, 确认报文段的 ACK 置 1, 确认号为 $u+1$, 序号为 v 等于 B 前面已传输过的数据的最后一个字节的序号加 1。进入 CLOSE-WAIT (关闭等待) 状态。

3) A 收到 B 的确认后, 进入 FIN-WAIT-2 (终止等待 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。

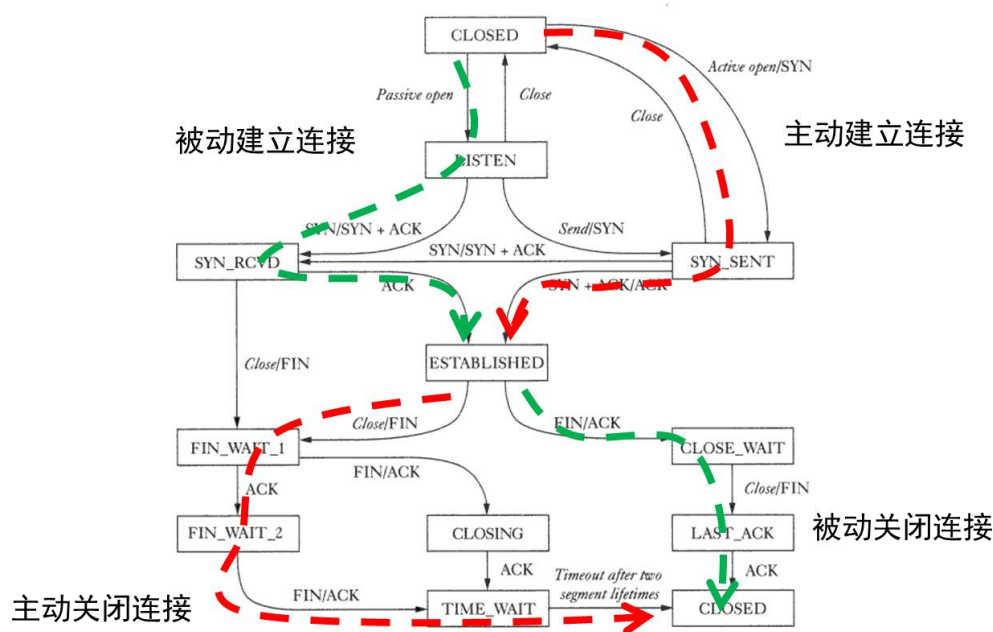
4) B 向 A 发送连接释放报文段, 该连接释放报文段的 FIN、ACK 置 1, 重复已发送过的确认号为 $u+1$, 进入 LAST-ACK (最后确认) 状态。

5) A 收到 B 的连接释放报文段后, 回复确认, 该确认报文段首部的 ACK 置 1, 序号为 $u+1$, 确认号为 $w+1$, 因为前面发送过的 FIN 报文段要消耗掉一个序号, 随后进入 TIME-WAIT (时间等待) 状态。

6) B 收到 A 的确认后, 释放 TCP sock, 进入 CLOSED 状态。释放子 sock。

7) A 必须经过时间等待计时器 (TIME-WAIT timer) 设置的时间 2MSL 后, 进入 CLOSED 状态。释放 TCP sock。

状态机:



图二 TCP 状态机

状态机的核心功能在于，每当主机收到一个 TCP 包，该包需要交付当前状态机所在状态的应答函数进行应答。下面我们来分析各个状态的应答函数：

当收到一个 TCP 包时，首先需要检测：1) 该包是否为空包 2) 该包校验和是否正确，如果有一个检测不通过就丢弃该包。

接下来，处理三个不需要接收数据的状态：CLOSED, LISTEN, SYN_RECV, 或者接收到 RST 报文。

CLOSED: 无论收到什么数据包，该状态下都不应相应，向发信方返回 RST 报文。

LISTEN: 监听状态下，如果收到的包不是 SYN，就返回 RST。

如果是 SYN，就准备连接：新建一个子 sock，初始化好相应项（四元项），将此 sock 添加到父 sock 的 listen_queue 中，表示子 sock 还在等待建立连接，将子 sock 状态设置为 SYN_RECV, 发送 SYN|ACK 报文，调用 tcp_hash 将子 sock 加入对应的哈希链表。

SYN_SENT: 正常情况下，会收到 SYN|ACK 的包：此时需要修改状态为 ESTABLISHED，并且发送 ACK 报文，随后调用 wake_up 唤醒在 tcp_sock_connect 等待连接握手的 TCP sock。

异常情况下，就返回 RST 报文，退出。

接收到 RST 报文：调用 tcp_sock_close 关闭连接，释放 sock。

接下来处理可能会收到数据的状态：

首先检查收到 TCP 的序列号 seq 是否在接收窗口中，如果不在就丢弃。

然后检查收到的包是不是 SYN，如果是，说明是一个非法的 SYN，另一端发生混乱，就调用 tcp_sock_close 关闭连接，并发送 RST 报文。

然后开始处理各个状态：

SYN_RECV: 如果收到的包是 ACK, 说明三次握手的第三次完成, 连接建立, 将子 sock 从父 sock 的 listen_queue 中删除, 加入到 accept_queue 中, 设置状态为 ESTABLISHED, 随后调用 wake_up 唤醒因 tcp_sock_accept 而等待新连接三次握手完成的父 sock。

ESTABLISHED: 调用 tcp_update_window_safe 更新接收窗口。如果是 FIN 包, 将状态转换至 CLOSE_WAIT, 随后发送 ACK 报文。如果需要主动结束连接, 需要调用 tcp_sock_close, 会发送 FIN|ACK, 设置 sock 状态为 FIN_WAIT_1。

CLOSE_WAIT: 该步没有特殊的应对函数。但是当状态处于 CLOSE_WAIT 时, 说明对端连接已经半结束, 本端在处理好所有事物后, 需要主动调用 tcp_sock_close 关闭连接, 状态将跳转至 LAST_ACK。

LAST_ACK: 如果接收到 ACK 包, 将状态设置为 CLOSED, 连接关闭, 调用 tcp_unhash 将 sock 去哈希, 连接被动结束。

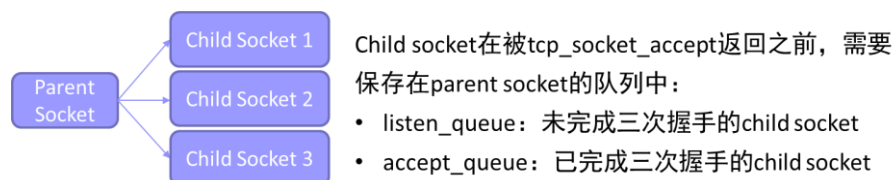
FIN_WAIT_1: 如果收到单纯的 ACK 包, 就跳转状态至 FIN_WAIT_2。如果收到 FIN|ACK 包, 就跳转到 CLOSING 状态, 并发送 ACK 包。

FIN_WAIT_2: 如果收到 FIN 包, 就跳转状态为 TIME_WAIT, 设置 timewait 计时器, 发送 ACK 包, tcp_unhash 将 sock 去哈希。

CLOSING: 如果收到 ACK 包, 就就跳转状态为 TIME_WAIT, 设置 timewait 计时器, 发送 ACK 包, tcp_unhash 将 sock 去哈希。

TIME_WAIT: 当某 sock 的计时器超时, 将 sock 从计序列中删除, 如果该 tsk 为父 sock, 需要执行 tcp_bind_unhash 去本地端口哈希, 设置状态为 CLOSED, 最终释放 sock, 连接主动结束。

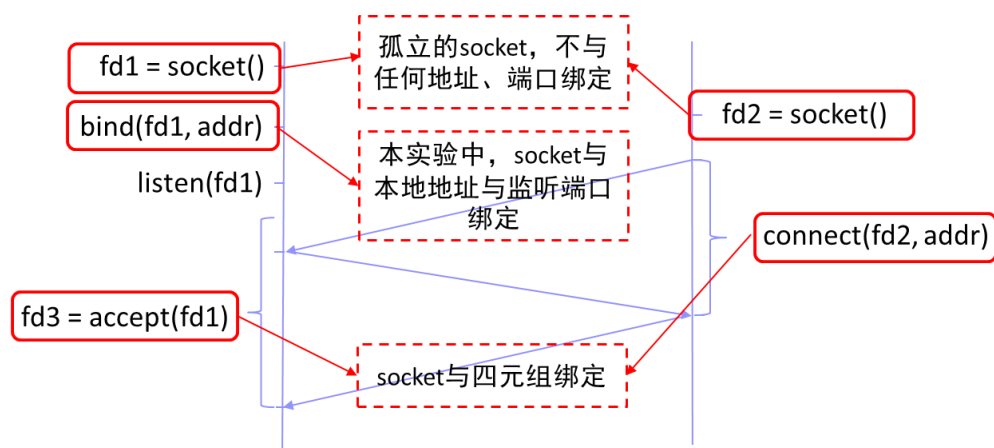
TCP 进行的核心元素为 **socket 的配置**, 每一个 socket 对应的一组 TCP 连接的操作。



图三 服务器一方的 socket

作为被动连接的一方 (服务器), 首先需要监听一个本地地址和本地端口, 需要建立一个 parent socket, 来响应任何到达该端口的请求信息, 之后调用 accept 操作陷入睡眠。每当有新的 TCP 的 SYN 请求到达该地址和端口时, 新建一个 child socket 来相应该请求, 而本 socket 不做动作, 继续监听该地址和端口, 以防有其他的 TCP SYN 连接。当有某个 child socket 三次握手成功, 就唤醒 parent socket, 把 child socket 提交到需要读取传输数据的应用程序处。如果解析新到达该地址和端口的 TCP 包, 发现该四元组与某一个 child socket 相匹配, 就使用该 socket 对其进行响应。连接结束后, child socket 将被删除。

作为主动连接的一方 (客户端), 就简单的多, 建立一个 parent socket, 直接调用 connect 操作与服务器进行相应即可。



图四 服务器（左）与客户端（右）socket 的绑定

socket 元组信息的绑定，如图所示。

服务器端，初始的 socket 不绑定任何信息。在进行监听之前，将 parent socket 绑定到本地地址与监听端口。当客户端发来连接请求后，经过 `accept` 操作，最终为此而建立的 child socket 与本连接的四元组进行绑定。

客户端，初始的 socket 不绑定任何信息。当客户端要与服务器进行交互时，在 `connect` 操作中，将 parent socket 与目标连接的四元组进行绑定。

socket 的查找：对于每一个 socket，通过遍历的线性方式效率是难以接受的。我们依旧采用 hash 查找。TCP 维护三个 hash 表：`established_table`、`listen_table` 和 `bind_table`。

1) 对于源目的地址、源目的端口都已经确定下来的 socket，按照上述 4 元组，将 `hash_list` 节点 hash 到 `established_table`

2) 对于只知道源地址、源端口的 socket，按照上述 2 元组，将 `hash_list` 节点 hash 到 `listen_table`

3) 任何占用一个本地端口的 socket，按照该端口号将 `bind_hash_list` 节点 hash 到 `bind_table`

4) 对于一个新到达的数据包，先在 `established_table` 中查找相应 socket，如果没有找到，再到 `listen_table` 中查找相应 socket

需要说明的是，只有真正占用了本地端口的 socket 才会进入 `bind_table`，比如说每个 parent socket。child socket 只是借用了其 parent socket 的端口。

socket 的结束等待：对于每一个主动关闭连接的 socket，会进入 `TIME_WAIT` 状态，该状态需要等待 2MSL 后，才能进入 `CLOSED` 状态。为实现该过程，对于每一个刚刚进入 `TIME_WAIT` 状态的 socket，我们将其挂载到停等计时链表上。计时器每过 MSL 时间会扫描一遍该链表，并将到达 2MSL 时间的 socket 从链表上删除，并彻底释放该 socket。

2. 启动脚本

1) TCP server client

```
make all
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# ./tcp_stack server 10001
h2# ./tcp_stack client 10.0.0.1 10001
mininet> quit
```

2) TCP 交互验证 server

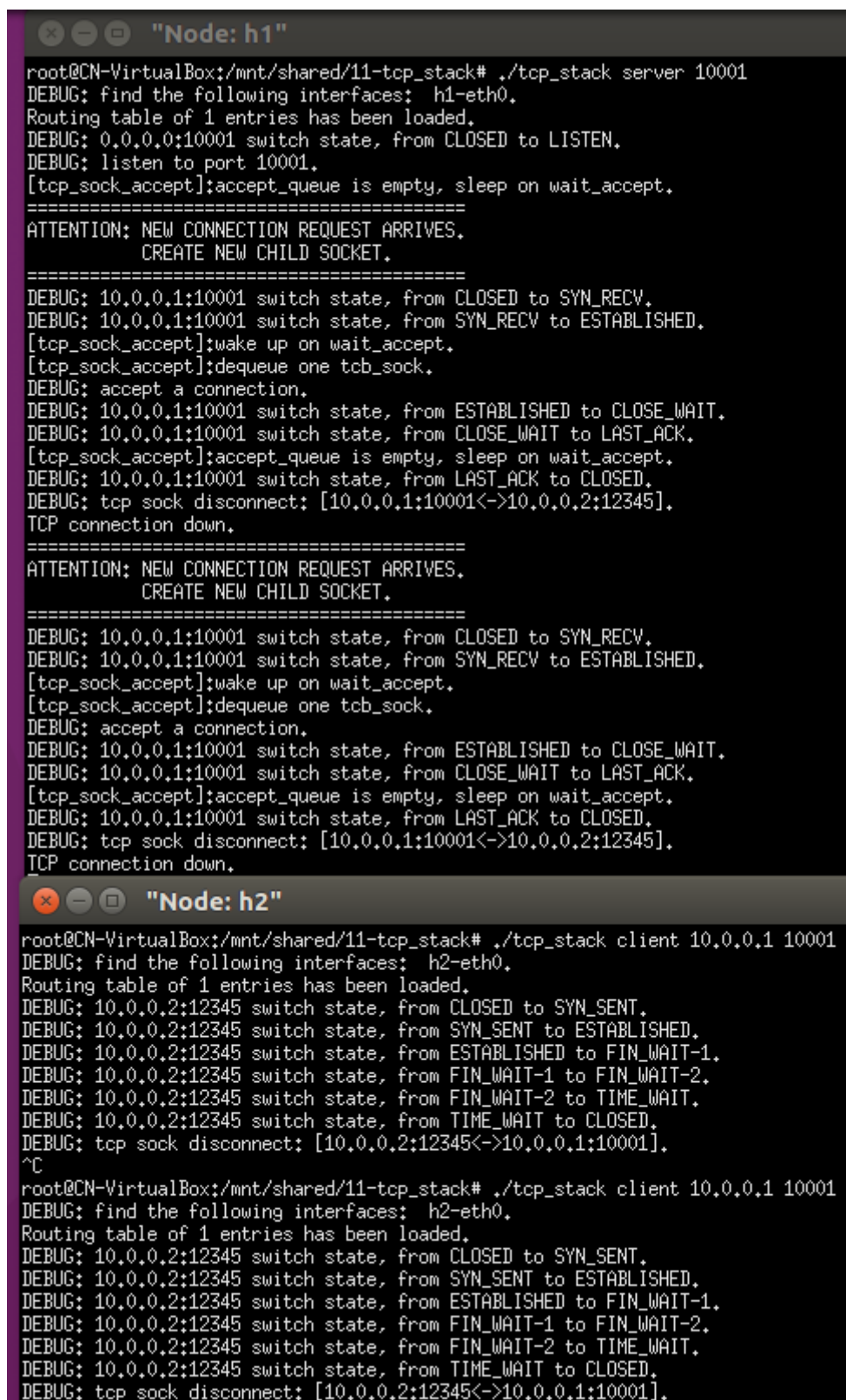
```
make all
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# wireshark
h1# ./tcp_stack server 10001
h2# python tcp_stack.py client 10.0.0.1 10001
mininet> quit
```

3) TCP 交互验证 client

```
make all
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# wireshark
h1# python tcp_stack.py server 10001
h2# ./tcp_stack client 10.0.0.1 10001
mininet> quit
```


三、实验结果及分析

1. 实验结果



The image shows two terminal windows. The top window, titled "Node: h1", shows the execution of a TCP server program. It starts by finding interfaces, loading a routing table, and listening on port 10001. It then receives two connection requests from 10.0.0.1:10001, each followed by a disconnect from 10.0.0.2:12345. The bottom window, titled "Node: h2", shows the execution of a TCP client program. It connects to 10.0.0.1:10001, and the logs show the state transitions from SYN_SENT to ESTABLISHED, then through FIN_WAIT-1, FIN_WAIT-2, and TIME_WAIT, finally disconnecting back to 10.0.0.2:12345. This sequence is repeated twice.

```

Node: h1
root@CN-VirtualBox:/mnt/shared/11-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.
[tcp_sock_accept]:accept_queue is empty, sleep on wait_accept.
=====
ATTENTION: NEW CONNECTION REQUEST ARRIVES.
CREATE NEW CHILD SOCKET.
=====
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
[tcp_sock_accept]:wake up on wait_accept.
[tcp_sock_accept]:dequeue one tcb_sock.
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.
[tcp_sock_accept]:accept_queue is empty, sleep on wait_accept.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: tcp sock disconnect: [10.0.0.1:10001<->10.0.0.2:12345].
TCP connection down.
=====
ATTENTION: NEW CONNECTION REQUEST ARRIVES.
CREATE NEW CHILD SOCKET.
=====
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
[tcp_sock_accept]:wake up on wait_accept.
[tcp_sock_accept]:dequeue one tcb_sock.
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.
[tcp_sock_accept]:accept_queue is empty, sleep on wait_accept.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: tcp sock disconnect: [10.0.0.1:10001<->10.0.0.2:12345].
TCP connection down.

Node: h2
root@CN-VirtualBox:/mnt/shared/11-tcp_stack# ./tcp_stack client 10.0.0.1 10001
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.
DEBUG: tcp sock disconnect: [10.0.0.2:12345<->10.0.0.1:10001].
^C
root@CN-VirtualBox:/mnt/shared/11-tcp_stack# ./tcp_stack client 10.0.0.1 10001
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.
DEBUG: tcp sock disconnect: [10.0.0.2:12345<->10.0.0.1:10001].

```

图二 TCP server client 实验结果

对于服务器端 h1:

起初开始时, 处于 CLOSED 状态。

随后由于主动开启监听端口 10001, 状态由 CLOSED 转移到 LISTEN, 此时由于调用 `tcp_sock_accept` 陷入等待 (因为此时没有连接被建立, 就打印 `sleep on wait_accept` 后睡眠)。

随后, 当监听端口传来的新的连接请求, 就新建一个子 `sock` 用来服务该连接, 设置子 `sock` 的状态为 `SYN_RECV`。

当对端的 ACK 到达后, 子 `sock` 状态转移到 `ESTABLISHED`, 此时唤醒 `tcp_sock_accept` (因为新的三次握手完成了, 打印 `wake up on wait_accept`。为提交该 `socket`, `accept` 操作将该 `socket` 从 `accept_queue` 出队, 打印 `dequeue one tcp_sock`, 进入 `server` 程序, 发出调试信息: `accept a connection`)。

之后收到了对端发来的 `FIN|ACK` 报文, 状态转移到 `CLOSE_WAIT`, 在自己发出 `FIN|ACK` 后, 转移到 `LAST_ACK`, 最后收到对端的 ACK 后转移到 `CLOSED`, 连接关闭, 释放子 `sock`。

对于客户端 h2:

起初处于 `CLOSED` 状态, 在发送 `SYN` 报文后状态转移到 `SYN_SENT`, 在收到 `SYN|ACK` 报文后转移到 `ESTABLISHED` 状态。

主动关闭连接, 发送 `FIN|ACK` 报文后转移到 `FIN_WAIT-1`, 收到了对端的 ACK 后转移到 `FIN_WAIT-2`, 再收到对端的 `FIN|ACK` 后转移到 `TIME_WAIT`, `TIME_WAIT` 超时后彻底关闭连接, 转移到 `CLOSED`, 释放 `sock`。

需要说明的是, 我对主程序和部分函数进行了修改, 实现了 TCP 的连续多次连接。测试结果中给出了连续两次连接服务器的请求, 结果令人满意。

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	3a:5c:83:d1:50:de	Broadcast	ARP	42	Who has 10.0.0.12 Tell 10.0.0.2
2	0.010878681	0e:c2:92:6c:70:c3	3a:5c:83:d1:50:de	ARP	42	10.0.0.1 is at 0e:c2:92:6c:70:c3
3	0.0210324983	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [SYN] Seq=0 Win=65535 Len=0
4	0.031324964	10.0.0.1	10.0.0.2	TCP	58	10001 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
5	0.041655544	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=1 Ack=1 Win=65535 Len=0
6	1.083678695	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=65535 Len=0
7	1.094817202	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [ACK] Seq=1 Ack=2 Win=29200 Len=0
8	5.054362467	10.0.0.1	10.0.0.2	TCP	54	10001 → 12345 [FIN, ACK] Seq=1 Ack=2 Win=29200 Len=0
9	5.064610106	10.0.0.2	10.0.0.1	TCP	54	12345 → 10001 [ACK] Seq=2 Ack=2 Win=65535 Len=0
10	73.316477113	3a:5c:83:d1:50:de	Broadcast	ARP	42	Who has 10.0.0.12 Tell 10.0.0.2
11	73.328199156	0e:c2:92:6c:70:c3	3a:5c:83:d1:50:de	ARP	42	10.0.0.1 is at 0e:c2:92:6c:70:c3
12	73.328215696	0e:c2:92:6c:70:c3	3a:5c:83:d1:50:de	ARP	42	10.0.0.1 is at 0e:c2:92:6c:70:c3
13	73.339902716	10.0.0.2	10.0.0.1	TCP	74	41914 → 10001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1
14	73.351138276	10.0.0.1	10.0.0.2	TCP	54	10001 → 41914 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
15	73.361403060	10.0.0.2	10.0.0.1	TCP	54	41914 → 10001 [ACK] Seq=1 Ack=1 Win=29200 Len=0
16	74.363462073	10.0.0.2	10.0.0.1	TCP	54	41914 → 10001 [FIN, ACK] Seq=1 Ack=1 Win=29200 Len=0
17	74.373667065	10.0.0.1	10.0.0.2	TCP	54	10001 → 41914 [ACK] Seq=1 Ack=2 Win=65535 Len=0
18	78.377061536	10.0.0.1	10.0.0.2	TCP	54	10001 → 41914 [FIN, ACK] Seq=1 Ack=2 Win=65535 Len=0
19	78.387281956	10.0.0.2	10.0.0.1	TCP	54	41914 → 10001 [ACK] Seq=2 Ack=2 Win=29200 Len=0

"Node: h2"

```

root@CN-VirtualBox:/mnt/shared/l1-tcp_stack# ./tcp_stack client 10.0.0.1 10001
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.
DEBUG: tcp sock disconnect: [10.0.0.2:12345<->10.0.0.1:10001].
^C
root@CN-VirtualBox:/mnt/shared/l1-tcp_stack# python tcp_stack.py client 10.0.0.1 10001
root@CN-VirtualBox:/mnt/shared/l1-tcp_stack#
```

"Node: h1"

```

root@CN-VirtualBox:/mnt/shared/l1-tcp_stack# python tcp_stack.py server 10001
root@CN-VirtualBox:/mnt/shared/l1-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.
[tcps_sock_accept]:accept_queue is empty, sleep on wait_accept.
=====
ATTENTION: NEW CONNECTION REQUEST ARRIVES.
CREATE NEW CHILD SOCKET.
=====
DEBUG: 10.0.0.1:10001 switch state, from CLOSED to SYN_RECV.
DEBUG: 10.0.0.1:10001 switch state, from SYN_RECV to ESTABLISHED.
[tcps_sock_accept]:wake up on wait_accept.
[tcps_sock_accept]:dequeue one tcp_sock.
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.
[tcps_sock_accept]:accept_queue is empty, sleep on wait_accept.
DEBUG: 10.0.0.1:10001 switch state, from LAST_ACK to CLOSED.
DEBUG: tcp sock disconnect: [10.0.0.1:10001<->10.0.0.2:41914].
TCP connection down.
```

图三 TCP 交互验证实验结果

交互验证，分别使用标准 TCP 代替自己写的客户端和服务端运行，进行抓包，发现两次实验中抓取数据包的类型相同，说明交互验证通过：

连接建立三次握手：

- h2 向 h1 发送 SYN 报文
- h1 向 h2 发送 SYN|ACK 报文
- h2 向 h1 发送 ACK 报文

连接释放四次握手：

- h2 向 h1 发送 FIN|ACK 报文
- h1 向 h2 发送 ACK 报文
- h1 向 h2 发送 FIN|ACK 报文
- h2 向 h1 发送 ACK 报文

(二) 实验代码详解

本次实验代码过长，不再赘述代码的内容。

一、 tcp_in.c:TCP 状态机具体实现

(1) void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet): 每当收到 TCP 数据包后，调用对应 socket 的信息响应数据包

当收到一个 TCP 包时，首先需要检测：1) 该包是否为空包 2) 该包校验和是否正确，如果有一个检测不通过就丢弃该包。

接下来，处理三个不需要接收数据的状态：CLOSED, LISTEN, SYN_RECV, 或者接收到 RST 报文。

CLOSED: 无论收到什么数据包，该状态下都不应相应，调用 tcp_send_control_packet 返回 RST 报文。

LISTEN: 监听状态下，如果收到的包不是 SYN，就调用 tcp_send_reset 返回 RST。

如果是 SYN，就准备连接：调用 alloc_tcp_sock 新建一个子 sock，设置四元项 sk_sip、sk_sport、sk_dip、sk_dport 为 control_block 的 daddr、dport、saddr、sport，新分配一个 iss，设置好 parent，rcv_nxt 为 cb->seq+1，将此 sock 添加到父 sock 的 listen_queue 中，表示子 sock 还在等待建立连接，将子 sock 状态设置为 SYN_RECV，调用 tcp_send_control_packet 发送 SYN|ACK 报文，调用 tcp_hash 将子 sock 加入对应的哈希链表。

SYN_SENT: 正常情况下，会收到 SYN|ACK 的包：此时需要修改状态为 ESTABLISHED，并且调用 tcp_send_control_packet 发送 ACK 报文，随后调用 wake_up 唤醒在 tcp_sock_connect 等待连接握手成功的 TCP sock。

异常情况下，就返回 RST 报文，退出。

接收到 RST 报文：调用 tcp_sock_close 关闭连接，释放 sock。

接下来处理可能会收到数据的状态：

首先调用 is_tcp_seq_valid，检查收到 TCP 的序列号 seq 是否在接收窗口中，如果不在就丢弃。

然后检查收到的包是不是 SYN，如果是，说明是一个非法的 SYN，另一端发生混乱，就调用 tcp_sock_close 关闭连接，并发送 RST 报文。

然后开始处理各个状态：

SYN_RECV: 如果收到的包是 ACK，说明三次握手的第三次完成，连接建立，将子 sock 从父 sock 的 listen_queue 中删除，加入到 accept_queue 中，设置状态为 ESTABLISHED，设置 rcv_nxt 为 cb->seq（注意没有+1!!!，因为该 ACK 报文不占用序列号），设置 snd_una 为 cb->ack，随后调用 wake_up 唤醒因 tcp_sock_accept 而等待新连接三次握手完成的父 sock。

ESTABLISHED: 调用 tcp_update_window_safe 更新接收窗口。如果是 FIN 包，将

状态转换至 CLOSE_WAIT, 设置 rcv_nxt 为 cb->seq+1, 随后发送 ACK 报文。如果需要主动结束连接, 需要调用 tcp_sock_close, 会发送 FIN|ACK, 设置 sock 状态为 FIN_WAIT_1。

CLOSE_WAIT: 该步没有特殊的应对函数。但是当状态处于 CLOSE_WAIT 时, 说明对端连接已经半结束, 本端在处理好所有事物后, 需要主动调用 tcp_sock_close 关闭连接, 状态将跳转至 LAST_ACK。

LAST_ACK: 如果接收到 ACK 包, 将状态设置为 CLOSED, 连接关闭, 调用 tcp_unhash 将 sock 去哈希, 连接被动结束。

FIN_WAIT_1: 如果收到单纯的 ACK 包, 就跳转状态至 FIN_WAIT_2。如果收到 FIN|ACK 包, 就跳转到 CLOSING 状态, 并发送 ACK 包。

FIN_WAIT_2: 如果收到 FIN 包, 就跳转状态为 TIME_WAIT, 调用 tcp_set_timewait_timer 设置 timewait 计时器, 发送 ACK 包, tcp_unhash 将 sock 去哈希。

CLOSING: 如果收到 ACK 包, 就就跳转状态为 TIME_WAIT, 设置 timewait 计时器, 发送 ACK 包, tcp_unhash 将 sock 去哈希。

TIME_WAIT: 当某 sock 的计时器超时, 将 sock 从计时间序列中删除, 如果该 tsk 为父 sock, 需要执行 tcp_bind_unhash 去本地端口哈希, 设置状态为 CLOSED, 最终释放 sock, 连接主动结束。

最后补充发送 ACK。

二、 tcp_timer.c:tcp_socket 定时器具体实现

(1) void tcp_scan_timer_list(): 扫描停等链表, 删除超时 socket

调用 list_for_each_entry_safe 遍历 timer_list, 将每个节点的 timeout 值减掉 TCP_TIMER_SCAN_INTERVAL, 如果 timeout 小于 0, 就将该节点从链表中删除, 并调用 timewait_to_tcp_sock 找到该计时节点所属的 socket, 如果该 socket 为 parent socket, 执行 tcp_bind_unhash 删除本地端口绑定。设置 socket 状态为 CLOSED, 释放 socket。

(2) void tcp_set_timewait_timer(struct tcp_sock *tsk): 将 socket 映射到停等链表

设置 timeout 为 TCP_TIMEWAIT_TIMEOUT, 调用 list_add_tail 将 timer 添加到 timer_list 上, 随后 ref_cnt 加 1, 表示该 socket 加入的链表数量增加 1。

三、 tcp_sock.c:tcp_socket 操作具体实现

(1) void free_tcp_sock(struct tcp_sock *tsk): 安全释放 socket 资源

先将 `ref_cnt` 减 1。如果减完后小于等于 0，说明没有该 socket 不在任何链表上，调用与 `alloc` 相反的操作：`free_ring_buffer`、`free_wait_struct`、`free_wait_struct`、`free_wait_struct`，最后释放 socket。

(2) `struct tcp_sock *tcp_sock_lookup_established(u32 saddr, u32 daddr, u16 sport, u16 dport)`：按照四元组查询 ESTABLISHED 表

首先调用 `tcp_hash_function(saddr, daddr, sport, dport)` 获得哈希值，根据哈希值遍历对应的 ESTABLISHED 哈希表，如果存在某个 socket 的四元组相匹配，就返回该 socket，否则返回 NULL。

(3) `struct tcp_sock *tcp_sock_lookup_listen(u32 saddr, u16 sport)`：按照源端口号查询 LISTEN 表

首先调用 `tcp_hash_function(0, 0, sport, 0)` 获得哈希值，根据哈希值遍历对应的 LISTEN 哈希表，如果存在某个 socket 的源端口号相匹配，就返回该 socket，否则返回 NULL。

(4) `int tcp_sock_connect(struct tcp_sock *tsk, struct sock_addr *skaddr)`：CONNECT 操作

首先调用 `tcp_get_port` 获取本地端口，随后将四元组 `sk_sip`、`sk_sport`、`sk_dip`、`sk_dport` 设置为获取的端口号、端口 IP、`ntohs(skaddr->port)`、`ntohs(skaddr->ip)`。注意，在终端输入的参数会先被转化成网络字节序保存到 `skaddr` 中，故调用时需要转成本地字节序。

随后调用 `tcp_bind_hash` 绑定本地端口。调用 `tcp_send_control_packet` 发送 SYN 包，设置状态为 `SYN_SENT`。最后调用 `tcp_hash` 处理 socket，调用 `sleep_on` 陷入等待。如果全部成功返回 0，否则返回 -1。

(5) `int tcp_sock_listen(struct tcp_sock *tsk, int backlog)`：LISTEN 操作

首先设置 `backlog` 为 `backlog`，设置状态为 LISTEN。

调用 `tcp_hash` 处理 socket，如果成功返回 0，否则返回 -1。

(6) `struct tcp_sock *tcp_sock_accept(struct tcp_sock *tsk)`：ACCEPT 操作

调用 `tcp_sock_accept_queue_empty` 判断 accept 队列是否为空。

如果为空，就调用 `sleep_on` 陷入等待。如果非空或者从睡眠中被唤醒，执行下面操作：调用 `tcp_sock_accept_dequeue` 返回一个子 socket。

(7) `void tcp_sock_close(struct tcp_sock *tsk): CLOSE 操作`

按当前 socket 所在状态进行响应：

LISTEN: 调用 `tcp_sock_clear_listen_queue` 清理监听对象，`tcp_unhash`，`tcp_bind_unhash`，设置状态为 CLOSED

SYN_RECV: 调用 `tcp_sock_clear_listen_queue` 清理监听对象

ESTABLISHED: 调用 `tcp_send_control_packet` 发送 FIN|ACK，跳转状态至 FIN_WAIT_1

CLOSE_WAIT: 调用 `tcp_send_control_packet` 发送 FIN|ACK，跳转至状态 LAST_ACK