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

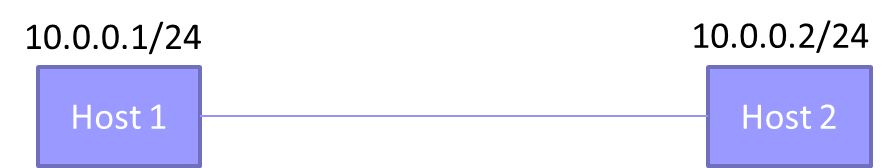
tcp\_out.c ： TCP发送相关函数

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



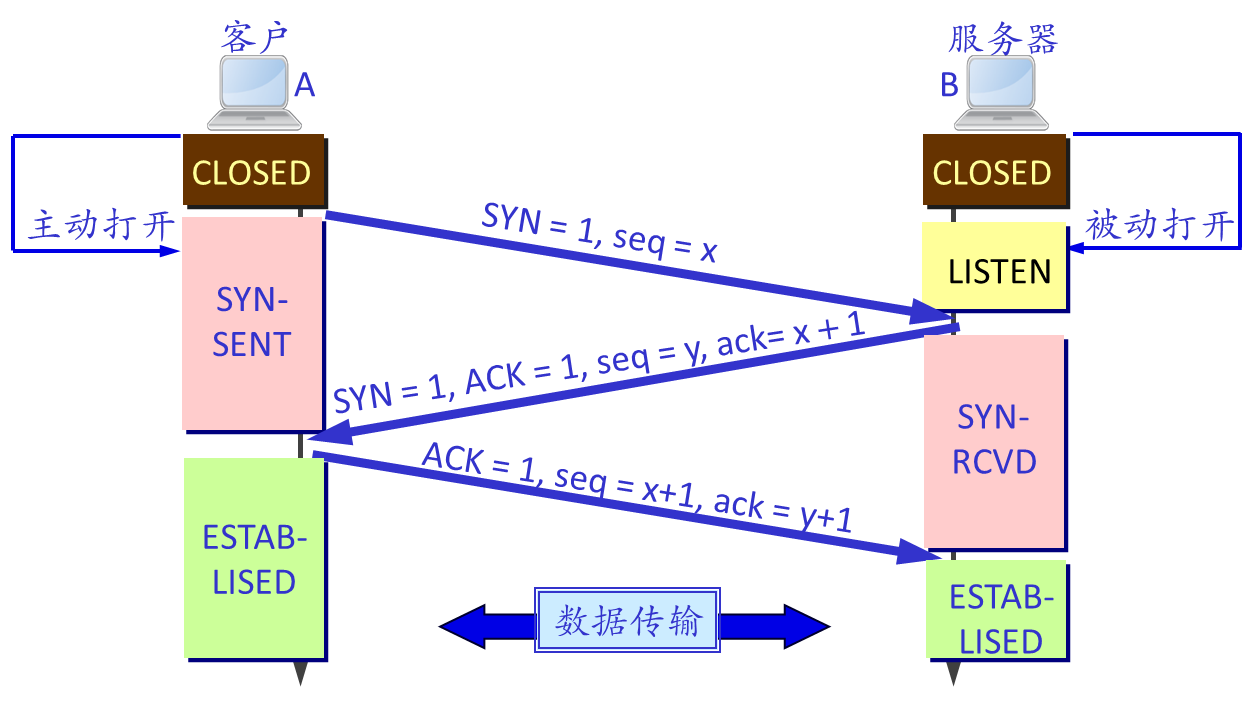
图一 二节点网络拓扑

TCP：Transmission Control Pro

tocol，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

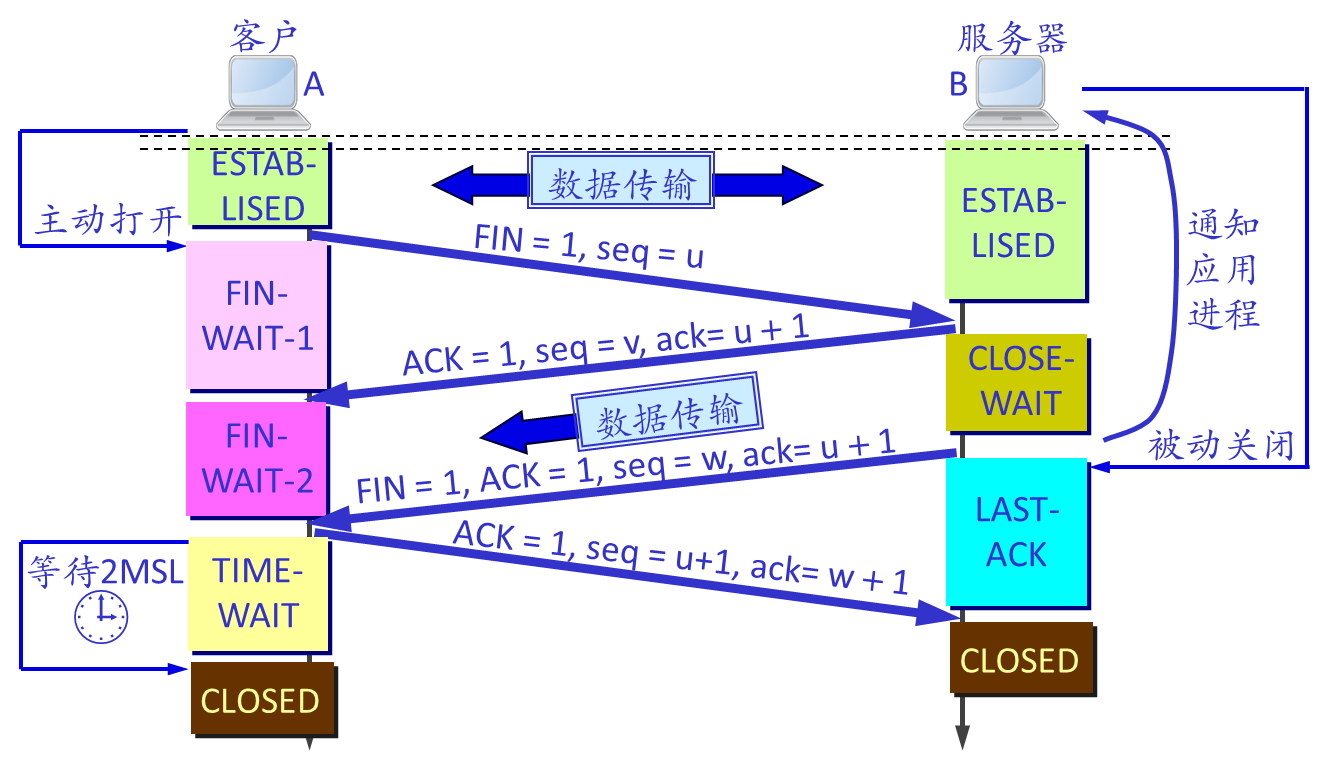
1）TCP客户进程A主动打开连接：创建TCP\_sock，向B发送连接请求报文段，请求报文段首部的同步位SYN置1，随机选择初始序号SequenceNum为x。随后进入SYN-SENT（同步已发送）状态，并陷入睡眠（connect操作），等待对端报文。

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Acknowlegment = 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。

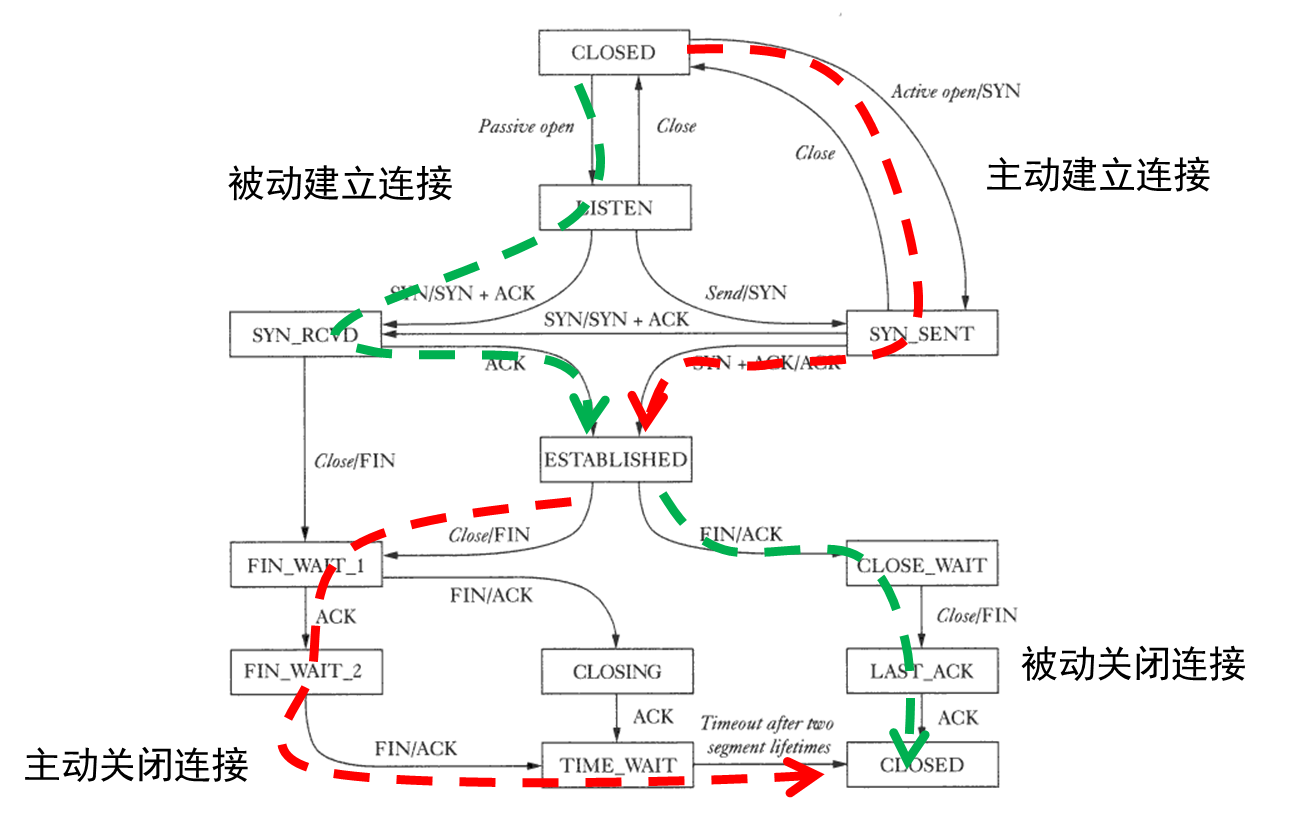
4）B向A发送连接释放报文段，该连接释放报文段的FIN、ACK置1，重复已发送过的确认号为 u+1，进入LAST-ACK (最后确认) 状态。

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。

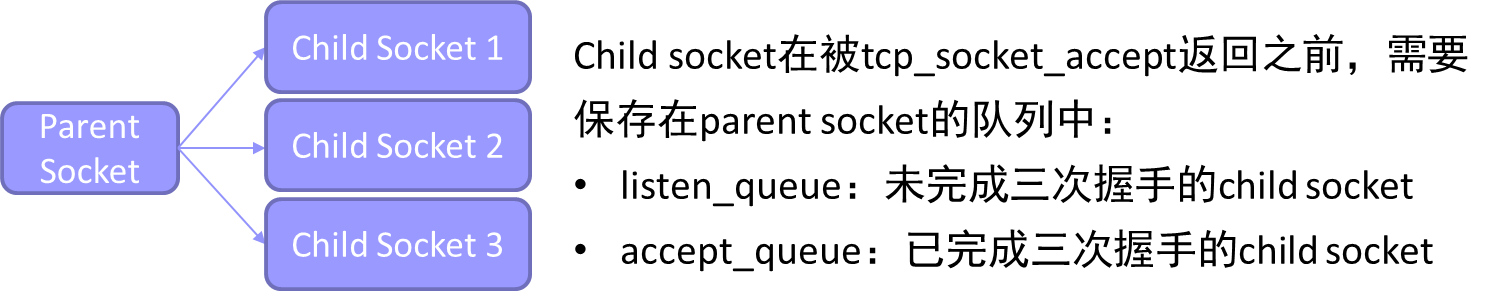
FIN\_WAIT\_1：如果收到单纯的ACK包，就跳转状态至FIN\_WAIT\_2。如果收到FIN|ACK包，就跳转到CLOSING状态，并发送ACK包。

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从计时序列中删除，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。

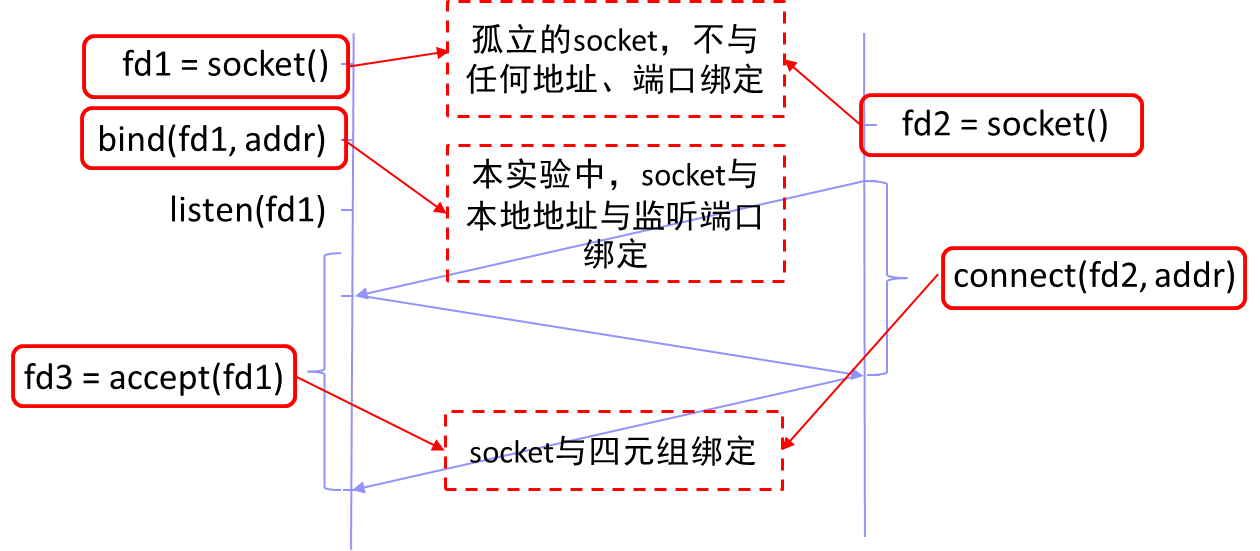
TCP进行的核心元素为socket的配置，每一个socket对应的一组TCP连接的操作。



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

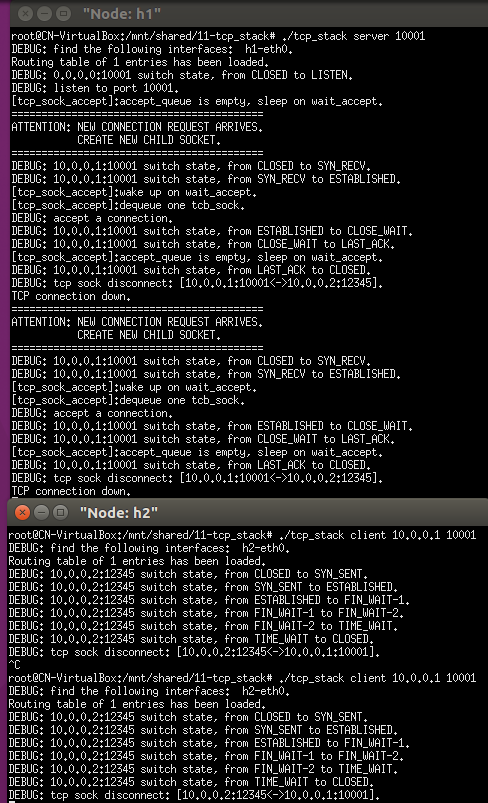
h1# python tcp\_stack.py server 10001

h2# ./tcp\_stack client 10.0.0.1 10001

mininet> quit

三、实验结果及分析

1. 实验结果



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）。

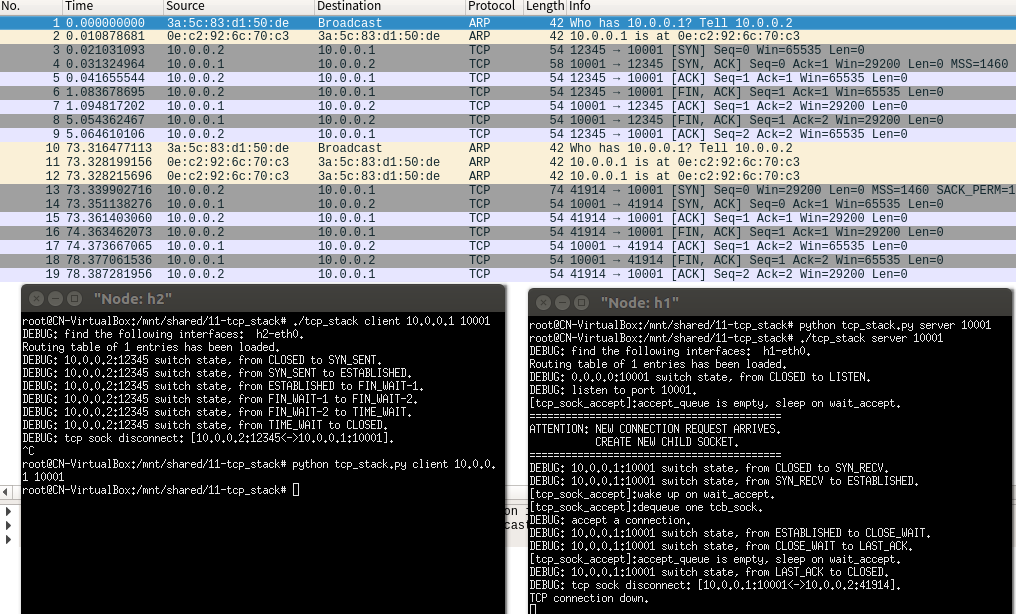
之后收到了对端发来的FIN|ACK报文，状态转移到CLOSE\_WAIT，在自己发出FIN|ACK后，转移到LAST\_ACK，最后收到对端的ACK后转移到CLOSED，连接关闭，释放子sock。

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。



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器运行，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。

1. 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，recv\_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。

1. 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。

1. 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、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，故调用时需要转化成本地字节序。

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