网络传输机制实验二报告

李昊宸

2017K8009929044

（一）TCP稳定传输实现

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h1并正确收发数据：client向server发送数据，server将数据echo给client

2. TCP server client文件传输实验：

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：Client发送文件client-input.dat给server，server将收到的数据存储到文件server-output.dat

4）比较两个文件是否完全相同

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 # 能够进行收发数据的tcp sock apps

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

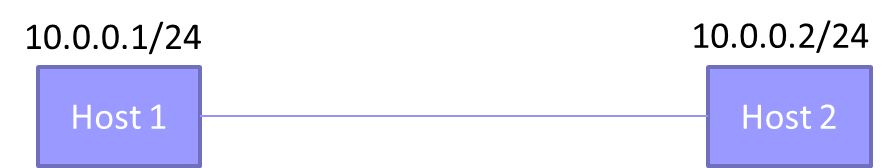
tcp\_sock.c ： tcp\_sock操作相关函数

tcp\_stack.py ： python应用实现，用于测试

tcp\_timer.c ： TCP定时器

create\_randfile.sh # 随机生成文件的脚本

tcp\_topo.py ： 实现二节点的简单拓扑



图一 二节点网络拓扑

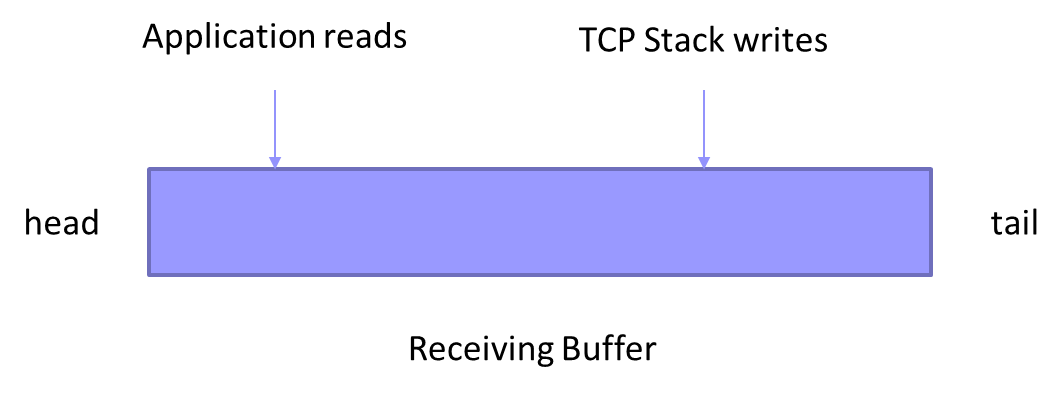
上周的实验已经实现TCP连接的建立和关闭，这周需要补充在连接建立后进行数据的传输。

TCP中数据接收和缓存：TCP协议栈收到数据包后，使用接收缓存来存储相应数据，供应用程序读取。

1）使用环形缓存(ring buffer)来实现

2）接收缓存大小为recv\_window

3）使用锁（pthread\_mutex\_t）来防止读写冲突



TCP数据发送流程：

1）待发送数据全部存储于上层应用buffer中

2）如果对端recv\_window允许，则发送数据

3）每次从buffer中读取1个数据包大小的数据，封装数据包，通过IP层发送函数，将数据包发出去

本实验中TCP两端时序图：

1）如果发送时，接收方（server）已经准备接收

|  |  |  |  |
| --- | --- | --- | --- |
| client  application | client  stack | server  stack | server  application |
|  |  |  | 之前调用tcp\_sock\_read  wait\_recv陷入睡眠 |
| tcp\_sock\_write  发送数据  wait\_send陷入睡眠 |  |  |  |
|  |  |  |  |
|  |  | tcp\_handle\_recv\_data获取recv\_buffer互斥锁 |  |
|  |  | 解析数据包，写recv\_buffer |  |
|  |  | 释放互斥锁 |  |
|  |  | 唤醒wait\_recv |  |
|  |  | 发送ACK报文 | tcp\_client 获取recv\_buffer互斥锁 |
|  | 更新发送窗口  唤醒wait\_send |  | 读取recv\_buffer |
| 准备下一次的发送 |  |  | 释放互斥锁 |
|  |  |  |  |

2）如果发送时，接收方（server）还在处理之前的接收信息

|  |  |  |  |
| --- | --- | --- | --- |
| client  application | client  stack | server  stack | server  application |
| tcp\_sock\_write  发送数据  wait\_send陷入睡眠 |  |  |  |
|  |  |  | tcp\_client 获取recv\_buffer互斥锁 |
|  |  | tcp\_handle\_recv\_data获取recv\_buffer互斥锁  失败，陷入睡眠 | 读取recv\_buffer |
|  |  |  | 释放互斥锁 |
|  |  | 获取互斥锁  解析数据包，写recv\_buffer |  |
|  |  | 释放互斥锁 |  |
|  |  | 发送ACK报文 |  |
|  | 更新发送窗口  唤醒wait\_send |  |  |
| 准备下一次的发送 |  |  |  |
|  |  |  |  |

本次实验的交互流程：

1. server端首先新建tcp\_socket，绑定指定本地地址和端口，对其监听，调用accept操作等待请求到达
2. client端首先新建tcp\_socket，调用connect操作请求server服务
3. server监听到请求，如同上周实验所做的，新建子tcp\_socket，与client请求进行交互，最后建立连接
4. client打开要发送的文件，将文件内容读取到缓冲区buffer中；server打开要保存的文件，调用tcp\_sock\_read操作等待要到达的数据（wait\_recv）
5. 重复以下操作：

client调用tcp\_sock\_write，将buffer中的部分数据发送出去，然后陷入等待（wait\_send）。

server接收到数据包后，调用tcp\_handle\_recv\_data处理数据：如果ring\_buffer满了，就需要丢弃该包（但是该实验还没有处理丢包的逻辑，所以这种情况需要控制它不会发生）；如果没满，就调用write\_ring\_buffer将能写入的接收数据写入到ring\_bufer中，唤醒tcp\_sock\_read（wait\_recv），将读到ring\_buffer中的数据读出，在应用内部做进一步的处理。之后修改rcv\_nxt和snd\_una的值，发送ACK报文。

client收到ACK报文后，会唤醒tcp\_sock\_write（wait\_send），表示发送完成。随后client会接着调用tcp\_sock\_read操作等待要到达的数据（wait\_recv）。

server接下来调用tcp\_sock\_write，将刚刚收到的消息带上“server echoes：”的字样全部发送回给client，然后陷入等待（wait\_send）。

client接收到数据包后，调用tcp\_handle\_recv\_data处理数据：如果ring\_buffer满了，就需要丢弃该包（但是该实验还没有处理丢包的逻辑，所以这种情况需要控制它不会发生）；如果没满，就调用write\_ring\_buffer将能写入的接收数据写入到ring\_bufer中，唤醒tcp\_sock\_read（wait\_recv），将读到ring\_buffer中的数据读出，在应用内部做进一步的处理。之后修改rcv\_nxt和snd\_una的值，发送ACK报文。之后睡眠1000微秒，等待下一轮重复。

server收到ACK报文后，会唤醒tcp\_sock\_write（wait\_send），表示发送完成。

最后，需要注意的是，收到数据包这一事件与两个过程相关：tcp\_sock\_read和tcp\_handle\_recv\_data。收到一个数据包，首先要经由tcp\_handle\_recv\_data处理。正常情况下，收到的数据可以全部放入环形buf中，每一次写完后都要唤醒一次wait\_recv（意义为如果有tcp\_sock\_read因环形buf为空陷入等待，就将其唤醒）。但是，如果出现了放不进去（buf满）的情况，就要陷入等待，等tcp\_sock\_read将数据从环形buf中读出去后，再次调用wake\_up唤醒tcp\_handle\_recv\_data，才能继续将未写入的数据写入。此时需注意，如果在等待写入期间，又有数据包到达，那么该包将丢失。

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 server client文件传输

make all

sudo python tcp\_topo.py

mininet> xterm h1 h2

h1# ./tcp\_stack server 10001

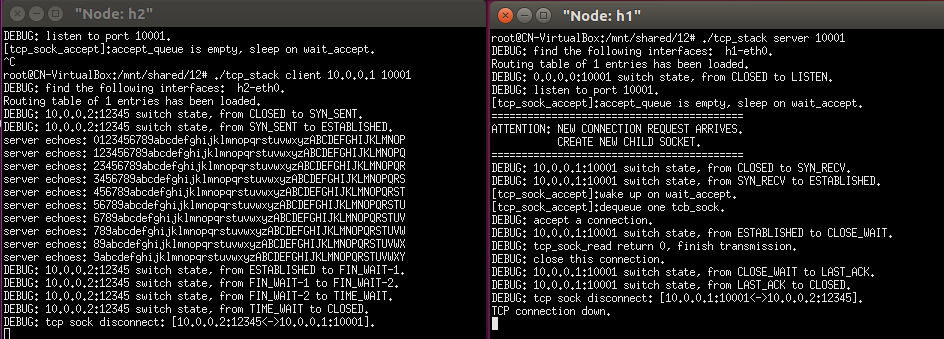
h2# python tcp\_stack.py client 10.0.0.1 10001

mininet> quit

注：上次实验已经验证状态转移的鲁棒性，故本次不再交叉验证（而且改py文件还蛮

麻烦，就不改了233）

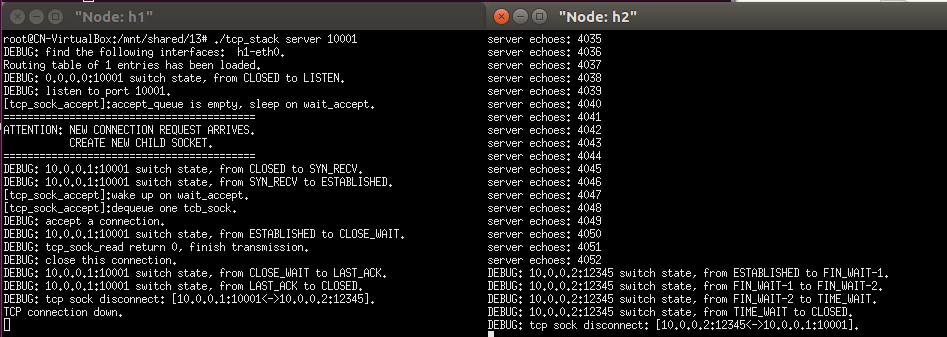
三、实验结果及分析

1. 实验结果

图二 TCP server client实验结果

可以看到，在三次握手连接建立完成后，client（h2）发送给h1的内容，被h1添加“server echoes：”报头后原封不动的返回了过来。传送起始点从“0”到“9”共更改10次，也体现在了打印结果上。

在预设的10次传输完成后，client（h2）发起关闭连接的请求，server（h1）响应之。整个状态变化过程与上一次实验相同。



图三 TCP server client文件传输实验结果

这次与之前有所不同的是，文件的内容是client（h2）打开本地的client\_input.dat文件读入的，大小为4052632B，以每个包大小为1000B发送给server（h1）。server（h1）每次收到数据后，将数据写入到本地的server\_output.dat中，并将计数器加1发送给client（h2）。在传输完成后，双方结束连接。



图四 TCP server client文件传输比较实验结果

调用diff命令比较client\_input.dat和server-output.dat，发现完全一致。说明实现了稳定传输。

（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)：状态响应函数

与上周作业相比，修改了状态响应函数：

如果当前状态为ESTABLISHED，并且收到数据包不是FIN包，意味着要进行数据处理：

如果收到的数据长度为0，说明只是ACK报文，表明自己是消息发送方，此时需要修改snd\_una为ack，rcv\_nxt为seq+1，并且调用tcp\_update\_window\_safe更新发送窗口。

否则，说明是数据报文，对方为发送方，此时调用tcp\_handle\_recv\_data：

首先进入一个循环，循环终止条件为收到的数据包中所有数据都写入到唤醒buf中：

检查唤醒buf是否已满。如果满，就调用sleep\_on（wait\_recv）陷入睡眠。如果未满/被唤醒，就向唤醒buf中写入数据（如果能全部写入就全部写入，如果不能就将环形buf写满），更新已经写入的数据量，调用wake\_up(wait\_recv)唤醒tcp\_sock\_read。

修改recv\_nxt为seq+len，snd\_una为ack。

最后发送ACK。

1. tcp\_sock.c:增加读写函数

（1）int tcp\_sock\_read(struct tcp\_sock \*tsk, char \*buf, int len)：从环形buf中读取数据到应用

首先进入循环：检查环形buf是否为空。如果空，就调用sleep\_on(wait\_recv)陷入睡眠，等待tcp\_handle\_recv\_data唤醒。如果非空，就调用read\_ring\_buffer读取数据，调用wake\_up(wait\_recv)唤醒tcp\_handle\_recv\_data，返回读取数据的长度。

（2）int tcp\_sock\_write(struct tcp\_sock \*tsk, char \*buf, int len)：将数据发送出去

进入循环，循环结束的条件为剩余待发送长度等于0：（考虑丢包重传）设置发送长度为：如果当前发送序号+待发送长度-发送确认序号大于默认窗口大小，就设置发送长度为默认窗口大小，否则设为当前发送序号+待发送长度-发送确认序号。调用tcp\_send\_data从发送确认序号标记处开始发送，发送刚刚计算的发送长度的数据，调用sleep\_on（wait\_send）陷入睡眠，等待被tcp\_update\_window唤醒，待发送长度减小发送长度的数值。

最后，如果顺利结束，返回len。

此处部分写的比较绕，主要要考虑的情况太多，仍有不完善的地方。等到下周实验开始写重传时再重新处理这些部分。