

计算机网 络研讨课

PPT报告



张磊 2017K8009922027



LAB 5

交换机转发实验

设计过程

1. 拷贝 lab 4 实验中的代码
完成 broadcast_packet 函
数的编写；

2. 完成 lookup_port 函数
的编写，查找应当从哪个
端口转发数据报；

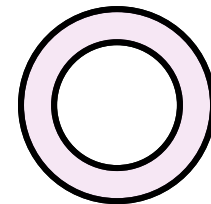
3. 完成 insert_mac_port 函
数的编写，插入新收到的
mac->port 映射；

5. 完成 handle_packet 函
数的编写；

6. 完成
sweep_aged_mac_port_en
try 函数的编写，定期检查
更新 mac->port 的映射；

7. 修改 main 函数，利用
pthread 库函数创建两个
线程分别执行数据 包转发
和转发表老化的操作；

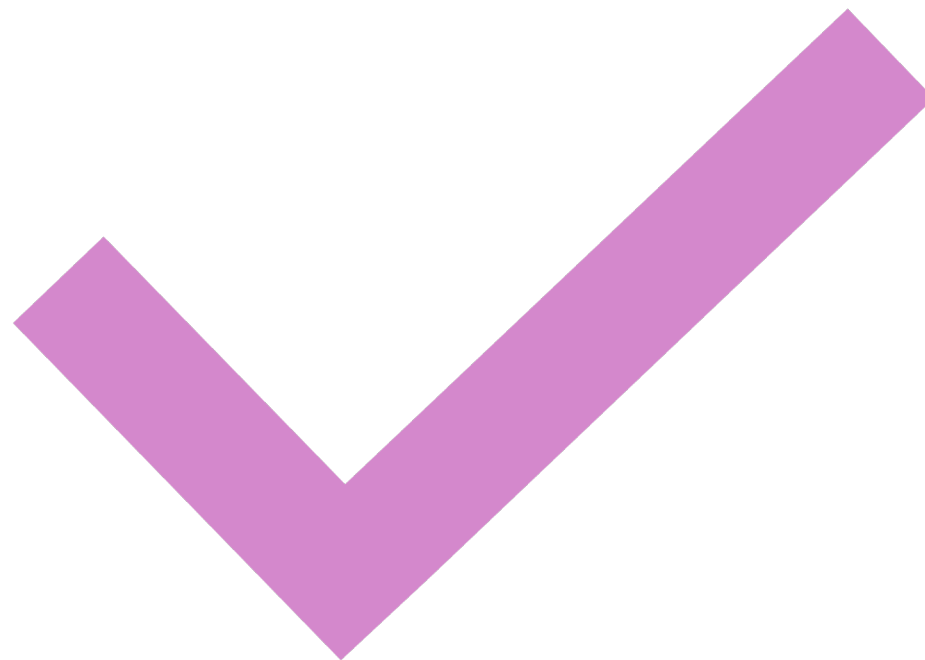
8. 使用 iperf 和给定拓扑进
行带宽测试；





遇到的问题

本次实验较为简单，实验完成的比较顺利，没有遇到什么大问题





结果分析

- 从实验结果中我们发现，当 h1 做 client 的时候，交换机转发的带宽 相比于集线器广播有明显的提升；而当 h1 做 server 的时候，交换机 转发的带宽与集线器广播相比变化不大；
- 前者的原因在于，h1 做 client 时，同时向 h2, h3 请求数据，h2 和 h3 通过交换机 s1 定向的将数据包发送给 h1 对应的端口，而对于集线器 b1，则会将来自 h2, h3 的数据通过广播的形式发送出去，也就是说，虽然 h2 到 h1 的数据包能够发送到 h1 但是多了一步将数据包发送到 h3 的操作，降低了有效带宽，因此交换机转发的效率有明显的提升；
- 后者的原因在于，h1 做 server 时，同时向 h2, h3 发送数据，虽然交换机 s1 发送的时候依然是定向发送，但是由于 h2, h3 都要数据，所以定向转发的效果与集线器广播的效果相同，因此测试带宽无明显变化；





反思总结

- 本次实验加深了我对交换机的理解，通过自己编写代码，我更加深入的理解的交换机的工作原理，使用交换机时需要注意的事项以及交换机与集线器的区别；
- 同时在思考题中，对于广播数据包和普通数据包对交换机的影响这个问题也让我对广播和定向转发这两个行为有了更加清楚的认识；



LAB 6

生成树机制实验

设计过程

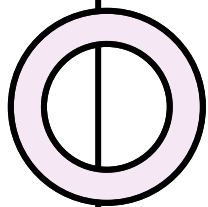
1. 基于附件中的代码，完成 stp.c 中对 stp_handle_cnfig_packet 函数的编写；

2. 运行 four_node_ring.py 拓扑，4 个节点分别运行 stp 程序，将输出重定向到 b*-output.txt 文件；

3. 等待一段时间，执行 pkill -SIGTERM stp 命令强制所有 stp 程序输出最终状态并退出；

4. 执行 dump_output.sh 脚本，输出 4 个节点的状态；

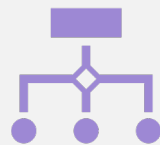
5. 按照 four_node_ring.py 的规则，编写 seven_node_ring.py 文件，重复上述实验；



遇到的问题



Four_node_ring 实验进展顺利，没有遇到大问题



在自己构造 topo 时，由于不熟悉 mininet，导致我在分配端口和IP 时出现了错误，起初我还以为是我写的代码有问题，后来想了一个晚上才找到问题



结果分析

- 1. Four_node_ring 的实验结果显示，stp 程序运行成功，成功去除原始环路中的冗余边，生成了最小生成树；
- 2. 在 seven_node_ring 实验中，我增加了链路的复杂度，构造了 7 个节点，4 条冗余边的环路，实验结果显示 stp 程序运行成功，成功构造出了这 7 个节点最小生成树；





反思总结

- 1. 本次实验的原理与 zookeeper 这类分布式服务器的分布式一致性协议非常相似，由于上学期在面向对象编程课程中选择了阅读 zookeeper 的源码，所以对这次实验的生成树算法比较容易理解，并且通过这次实验，也让我复习了上学期阅读 zookeeper 源码的很多收获；
- 2. 通过这次实验，我对计算机网络协议中的生成树协议算法的运行机制又有了更进一步的理解，果然只有配合实验，才能更好的对理论课上学到的知识进行消化和吸收；





LAB 7

路由器转发实验

设计过程

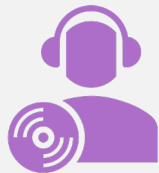
1. 基于附件中的代码，完成 arp.c, arpcache.c, icmp.c, ip_base.c, ip.c 的编写，实现路由器对数据包的转发处理功能；

2. 运行 router_topo.py 拓扑，在 r1 节点上运行 router 程序，在 h1 上进行 ping 实验；

3. 自己编写包含多个路由器节点的 topo 文件，手动配置其默认路由表，完成连通性测试和路径测试；



遇到的问题

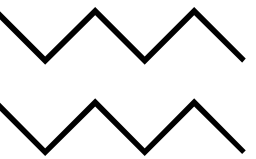


本次实验代码量较之前的实验有了巨大的提升，耗费时间比较长



在编写 icmp.c 函数时，由于没有搞清楚 icmp 数据报的头部和数据部分的分解，导致回复 icmp 数据报时校验和错误





结果分析

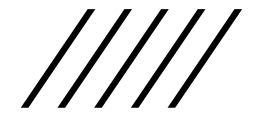
实验一，成功在 h1 上 ping 通 r1, h2, h3；并且在 ping 10.0.3.11 时，如期返回 ICMP Destination Host Unreachable；在 ping 10.0.4.1 时，如期返回 ICMP Destination Net Unreachable；


实验二，构造 TOPO 满足路由器节点数要求(3 个)和主机节点跳数要求(至少 3 跳),并在 h1 上成功 ping 通 h2，ping 通 r1, r3, r2，完成连通性测试，traceroute 成功显示路径节点 IP 信息，完成路径测试；





反思总结

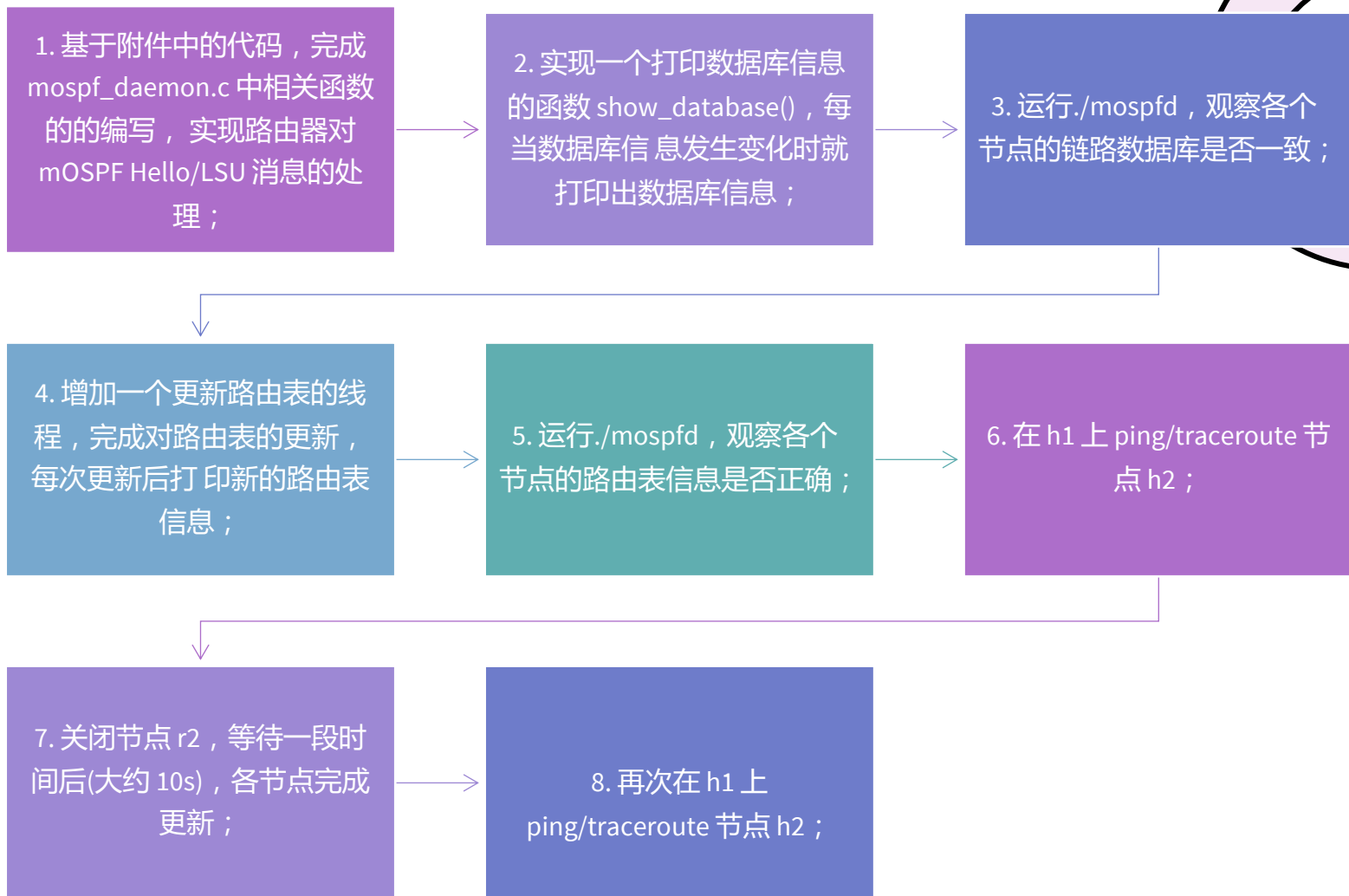
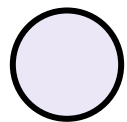
- 1. 本次实验代码量较之前的实验有了巨大的提升，所以耗费时间比较长，但是通过编写代码和 DEBUG，让我对路由器如何进行数据包的转发，对 ICMP 报文的格式，ARP 报文的格式都有了更加深刻的记忆和理解；
 - 2. 通过对 arp.c, arpcache.c 的编写，让我对路由器如何利用 arp 协议进行数据包转发，以及 IP 地址和 MAC 地址在网络传输中的作用有了更加深刻的认识，MAC 地址只在局域网内起作用，而 IP 在整个网络中都起作用；
 - 3. 在 ip.base.c, ip.c, icmp.c 的编写中，加深了我对路由器在网络层是如何处理收到的各种数据包的理解；
- 

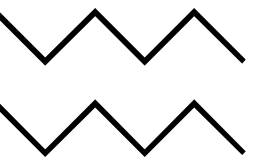


LAB 8

网络路由实验

设计过程





遇到的问题

本次实验代码量较大，
耗费时间较长

一开始不明白 lsu 的意
义，所以一直不知道怎
么下手，最后向同学请
教后，茅塞顿开

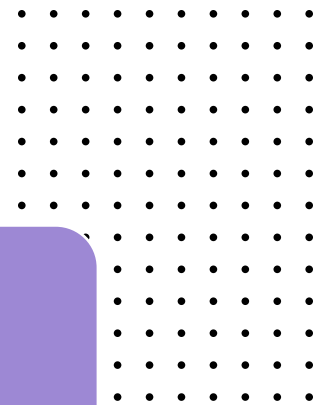


结果分析

1. 实验一中,我们观察到各个节点都形成了一致的链路数据库,实验内容一成功;

2. 实验二中,我们观察到各个节点都生成了正确的路由表项,在 h1 上 ping 节点 h2 时,可以 ping 通;在 h1 上 traceroute 节点 h2 时,可以看到消息依次经过了 r1,r2,r4 到达 h2;

3. 在关闭 r2 之后,等待一段时间,各个节点完成链路数据库和路由表项的更新,在此在 h1 上 ping 节点 h2,成功 ping 通,并且在 h1 上 traceroute 节点 h2 时,发现路由路径由原来的 h1->r1->**r2**->r4->h2 改变为了现在的 h1->r1->**r3**->r4->h2,实验二内容成功;





反思总结



1. 本次实验代码量较还是很大，并且在使用 Dijkstra 算法计算最短路径时比较麻烦，所以耗费时间比较长，但是通过编写代码和 DEBUG，让我对路由器如何交换路由信息，自动生成路由表有了更加深刻的理解，对 Hello 和 LSU 两种 mOSPF 消息的格式都有了更加深刻的记忆和理解；

2. 在进行路由表的更新的时候我尝试了两种操作，一种是在完成 handle_lsu_packet 之后更新路由表，这样做的好处是每次路由信息发生变化都能及时的更新路由表，缺点是，当网络中的节点数量变多后，更新路由表的信息的操作会变得非常频繁；

3. 因此我采用了第二种方法，单独创建一个用于更新路由表的线程，每隔一段时间（实验中我设置为 10 秒）更新一次路由表，这样做的好处是，即使网络中的节点很多，也不会频繁的更新路由表，因为我们知道，网络中链路很少会在极短时间内发生变化，如果每次收到 LSU 消息都去更新一次路由表的话，很多都是无效的操作，因为路由表并没有产生变化，这样做的缺点是可能无法及时的更新路由表中变化的信息，但是这个可以通过手动配置刷新时间来改变；



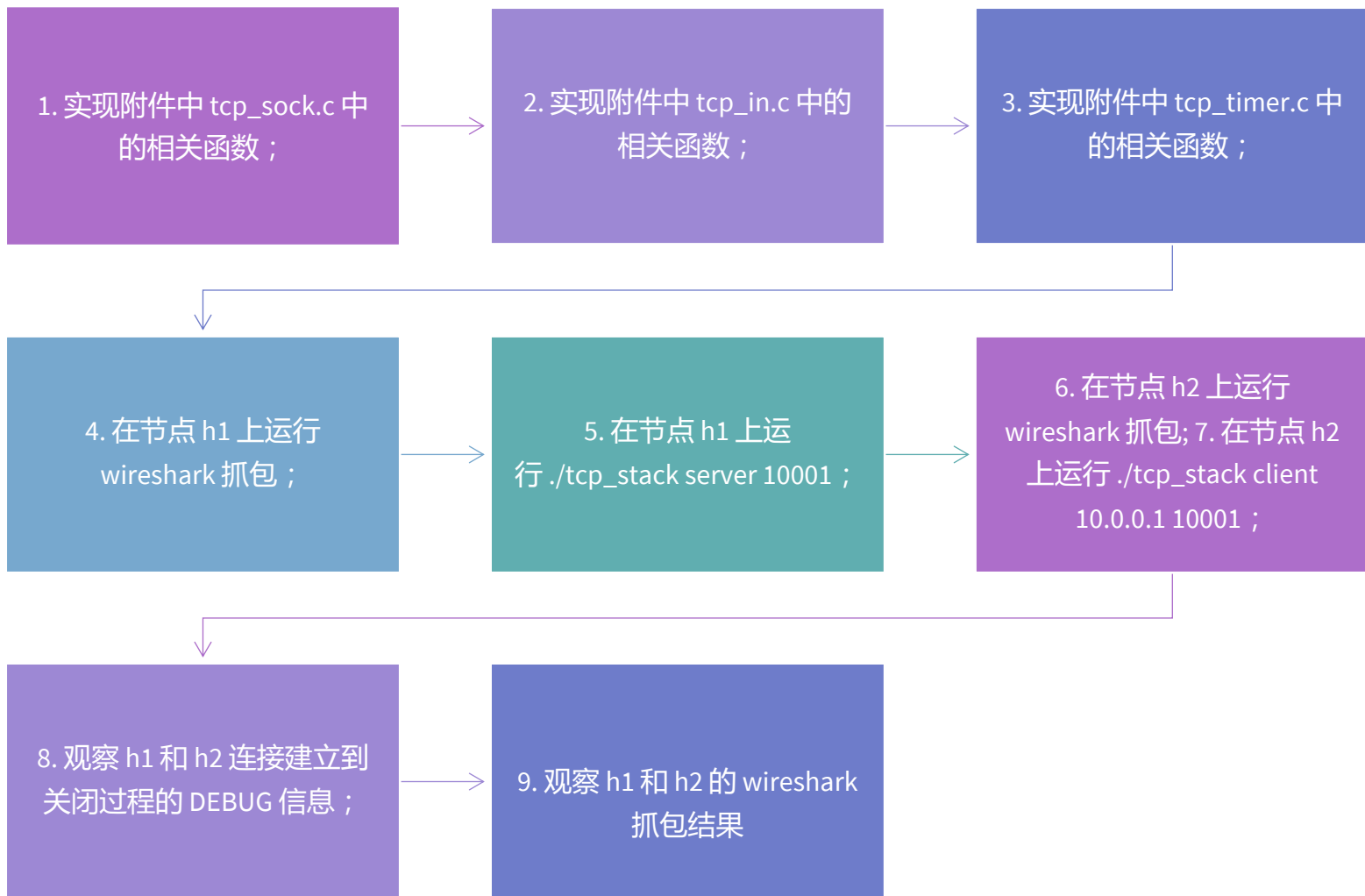
LAB 11

TCP 网络传输机制

实验一



设计过程





遇到的问题

本次实验较为繁琐，一开始没有找到头绪，先从 main 函数开始看了一遍代码，和需要实现的函数，然后又看了一次老师的讲解视频，然后又看了一遍理论上讲解的 TCP 的内容，最后晚上睡觉的时候过了一遍内容，才把条理弄清楚；

开始时，对于如何分配 socket，初始化 socket 不是很清楚，导致前期进展缓慢，之后的内容实现起来就比较顺利了






结果分析

1. 首先，通过观察 h1 终端输出的 DEBUG 信息，我们可以看到，h1 作为 TCP 栈的 server 端，依次经历了 CLOSED，LISTEN，SYN_RECV，ESTABLISHED，CLOSE_WAIT，LAST_ACK，CLOSED 这几个状态，成功的建立并关闭 TCP 连接；

2. 观察 h2 终端输出的 DEBUG 信息，我们可以看到，h2 作为 TCP 栈的 client 端，依次经历了 CLOSED，SYN_SENT，ESTABLISHED，FIN_WAIT_1，FIN_WAIT_2，TIME_WAIT，CLOSED 这几个状态，成功的建立并关闭了 TCP 连接；

3. 通过观察 h1 和 h2 的抓包结果，我们看到，h2 首先向 h1 发送了一个 syn 包，随后 h1 向 h2 回复 syn 和 ack 包，h2 收到后向 h1 发送 ack 包，至此，h1 和 h2 之间完成了 TCP 连接的建立；4. 随后，h2 向 h1 发送 fin 包，h1 收到 fin 包后向 h2 回复 ack，再然后 h1 向 h2 发送 fin 包，h2 收到 fin 包后向 h1 回复 ack 包，h1 收到来自 h2 的 ack 包，至此，h1 和 h2 完成 TCP 连接的关闭；

反思总结



1. 感觉这次实验和之前写的 CPU 有点像，都是根据状态机状态进行相应的处理，本次实验的几个状态非常完美的讲 server 端和 client 端的状态分开了(除了 CLOSE 状态)，这样实现起来就方便很多了；

2. 本来一开始，我是想先根据收到的包是 syn 包，还是 fin 包，还是 ack 包，再用不同状态来确定具体操作，但是看到 tcp_process 上老师给的注释是根据状态来区分，于是就改了，后来发现，还是根据状态来确定做些什么逻辑更加清晰；

3. 通过这次实验，让我对 TCP 的连接的建立和关闭有了更加清晰和深刻的理解，之前一直不太清楚的三次握手协议现在也搞清楚了；



LAB 12

TCP 网络传输机制

实验二



设计过程

实验内容一

1. 在tcp_sock.c中实现
tcp_sock_read函数和
tcp_sock_write函数

2. 在tcp_in.c中增加对收取数
据包的处理

3. 在节点h1上运
行 ./tcp_stack server 10001 启
动TCP协议栈服务器模式

4. 在节点h2上运
行 ./tcp_stack client 10.0.0.1
10001 启动TCP协议栈客户端
模式

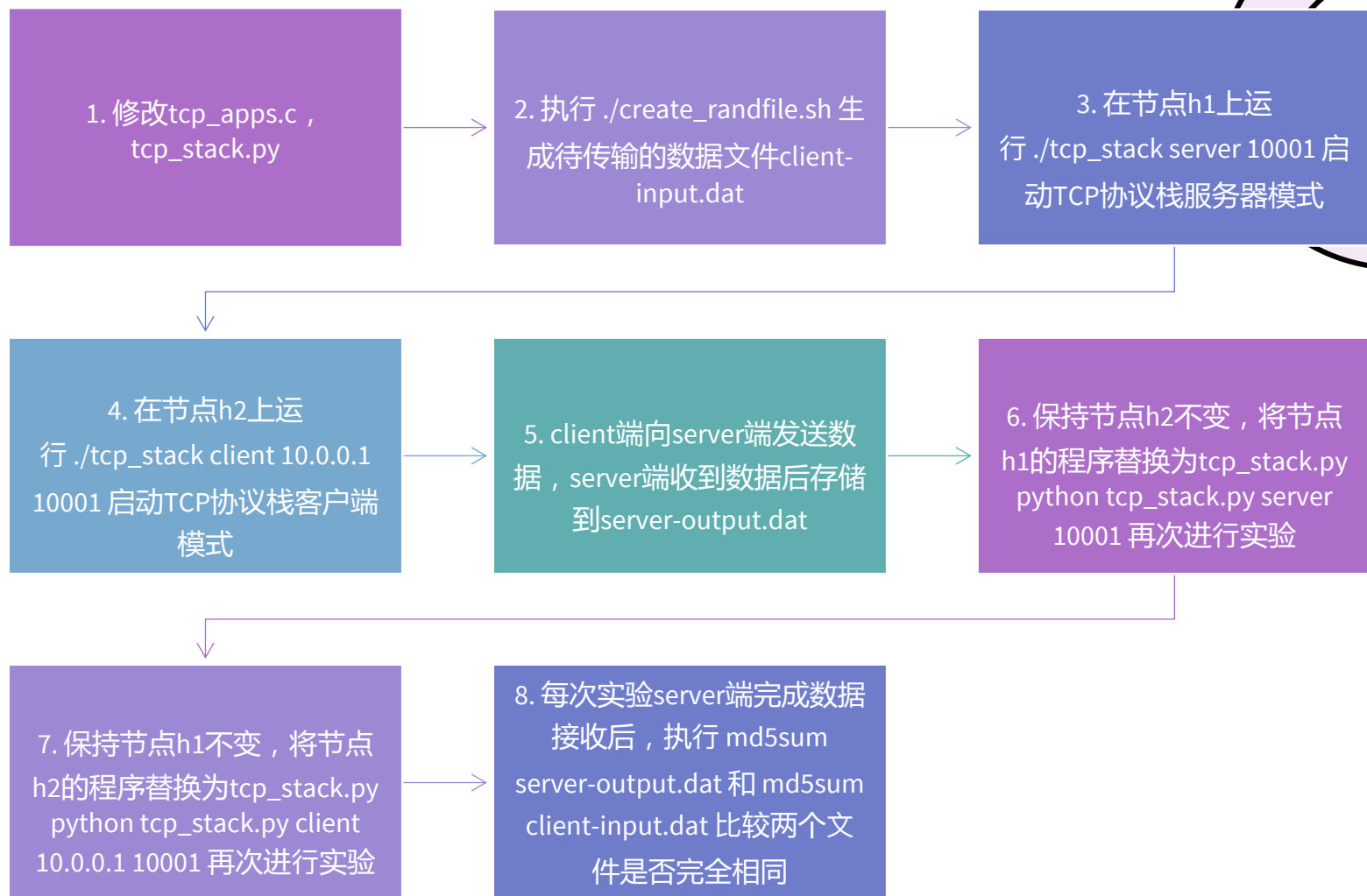
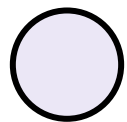
5. client端向server端发送数
据，server端收到数据后echo
给client端

6. 保持节点h2不变，将节点
h1的程序替换为tcp_stack.py
python tcp_stack.py server
10001 再次进行实验

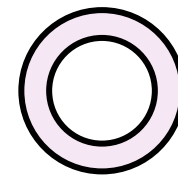
7. 保持节点h1不变，将节点
h2的程序替换为tcp_stack.py
python tcp_stack.py client
10.0.0.1 10001 再次进行实验

设计过程

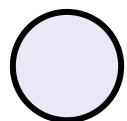
实验内容二



遇到的问题



这次实验本来很简单，但是由于在上一个实验中，对多进程访问相同数据时的处理不够严谨，导致这个实验中多出了许多bug



结果分析

实验内容一

1. 观察实验结果，当 h1 和 h2 都使用我们自己编写的 tcp_stack.c 文件时，client 端成功发送数据，server 端成功接收到 client 端发送的数据并 echo 给了 client 端，并成功关闭 TCP 连接，实验成功

2. 观察实验结果，h1 使用 tcp_stack.py 程序，h2 使用 tcp_stack.c 程序，client 端成功发送数据，server 端成功接收到 client 端发送的数据并 echo 给了 client 端，并成功关闭 TCP 连接，实验成功

3. 观察实验结果，h1 使用 tcp_stack.c 程序，h2 使用 tcp_stack.py 程序，client 端成功发送数据，server 端成功接收到 client 端发送的数据并 echo 给了 client 端，并成功关闭 TCP 连接，实验成功

4. 当 h1 使用 tcp_stack.py 程序时，最后需要收到一个空数据包才能退出循环，这里我直接使用 ctrl+c 中断程序执行

结果分析

实验内容二

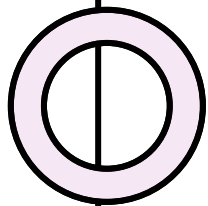
1. 观察实验结果，当 h1 和 h2 都使用我们自己编写的 tcp_stack.c 文件时，client 端成功发送数据，server 端成功接收到 client 端发送的数据并存储到 serveroutput.dat 文件中，并成功关闭TCP连接，使用 md5sum 程序检测收到的文件和发送的文件，结果相同，证明两个文件完全相同，实验成功

2. 观察实验结果，h1 使用 tcp_stack.py 程序，h2 使用 tcp_stack.c 程序，client 端成功发送数据，server 端成功接收到 client 端发送的数据并存储到 serveroutput.dat 文件中，并成功关闭 TCP 连接，使用 md5sum 程序检测收到的文件和发送的文件，结果相同，证明两个文

3. 观察实验结果，h1 使用 tcp_stack.c 程序，h2 使用 tcp_stack.py 程序，client 端成功发送数据，server 端成功接收到 client 端发送的数据并存储到 serveroutput.dat 文件中，并成功关闭 TCP 连接，使用 md5sum 程序检测收到的文件和发送的文件，结果相同，证明两个文

4. 当 h1 和 h2 都使用 tcp_stack.c 文件和 h1 使用 tcp_stack.py 程序，h2 使用 tcp_stack.c 程序时，客户端可以以较快速率发送数据

5. 当 h1 使用 tcp_stack.c 文件，h2 使用 tcp_stack.py 程序时，发送速率较低，且每次最多只能发送 500bytes 的数据，实验中，我使用的是 400bytes，当一次发送的数据量超过 500bytes 时，或者发送速率过快时会产生丢包，主要现象是 server端可能会出现只能读取 456byets 数据的情况



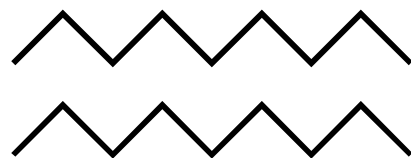
反思总结

1. 本次实验代码量较小，需要实验的内容也比较简单，但是还是碰到了一些比较刁钻的 bug，比如在 ring_buffer 结构体中添加了 lock 变量为互斥锁后，忘记初始化 lock，导致后续的实验中进程卡死，耗费了较多时间；

2. 在实验二中，可能是由于最后的 server 进程没有正确执行到 tcp_close 和 fclose，导致我的最后一部分接收到的数据存留在缓冲区，而没有写入到 server-output.dat 文件中，耗费了我一个下午的时间，最后请老师帮忙查看，在 fwrite 函数后执行 fflush 函数将缓冲区的数据写入到 server-output.dat 中，成功解决了这个问题

3. 在 DEBUG 的时候，由于怀疑 server 程序有问题，所以在每次接受时加了一个 sleep 函数，结果导致可能会出现第一个发送的数据包丢失的现象出现

4. 当需要 sleep 小数秒时间的时候，需要采用 usleep 函数，如果采用 sleep 函数，貌似对睡眠时间小于 1 的处理是直接归零了，这也导致我一开始对发送速率的错误控制



**THANK
YOU**

