**网络实验报告 exp1**

**刘耕印 2020K8009929032**

1. **互联网协议实验**
2. 执行wget与抓包

启动mininet，创建h1，由h1对[www.ucas.ac.cn发起wget](http://www.ucas.ac.cn发起wget)请求，使用wireshark进行抓包。请求成功，如下图所示：

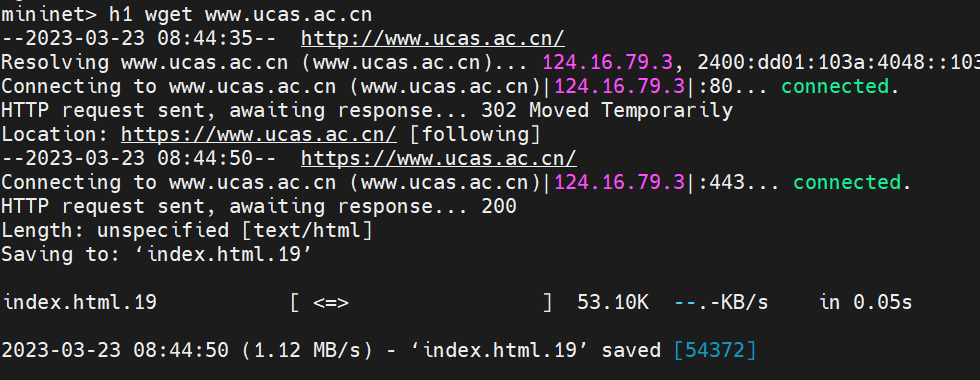


Figure h1申请成功，数据保存到当前目录

此处h1的ip地址是10.0.0.1。

接下来我保存wireshark的抓包记录为pdf格式，在第2部分进行简要分析。

1. 初探互联网协议，并浅析wget过程

观察抓包记录发现，此次传输主要涉及DNS, TCP以及HTTP几种协议。同时，在传输之前观察到ARP协议。

* ARP协议是根据ip地址查询MAC地址的一种协议。ARP报文格式如下：

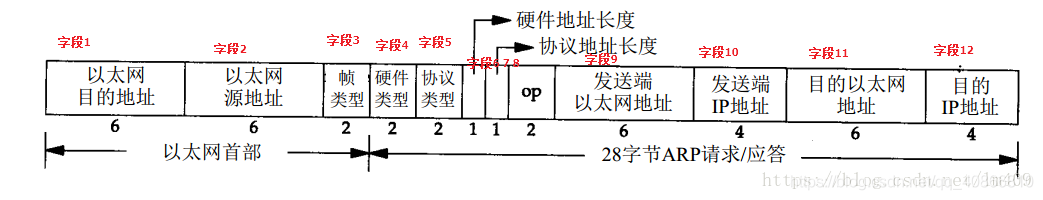


Figure ARP报文

发起的ARP请求如下：

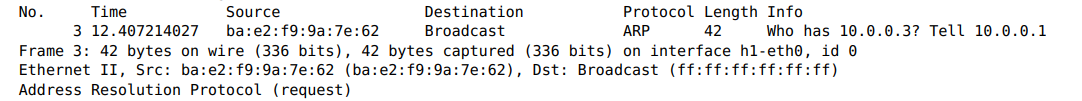


Figure ARP请求

此请求由ip10.0.0.1广播，目的是获取10.0.0.3的MAC地址。上述报文中“目的ip地址”设为10.0.0.3，以期应答。

收到的应答如下：

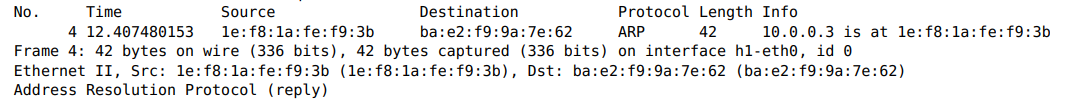


Figure ARP应答

此应答由ip10.0.0.3发送给10.0.0.1，上述报文中“发送端以太网地址”设为1e:f8:1a:fe:f9:3b，作为回答。如此一来，10.0.0.1（h1）就得到了10.0.0.3（linux虚拟主机）的MAC地址。至于h1为什么需要这个地址，我猜想是准备和虚拟机建立数据传输，但和本次wget动作关系不大。

在抓包记录的最后还出现了由10.0.0.3向10.0.0.1发送的单播ARP请求，其目的是确认10.0.0.1与其MAC地址间的映射是否发生改变，以维护10.0.0.3本地的ARP快取。这个请求收到回应，说明上述映射关系仍然有效。

* DNS协议是根据域名查询服务器ip地址的协议。客户查询服务器ip采用递归查询，将查询请求发送到本地DNS服务器（本例中为1.2.4.8），本地服务器在其记录中查找。若找到，则将查询结果返回给客户机。

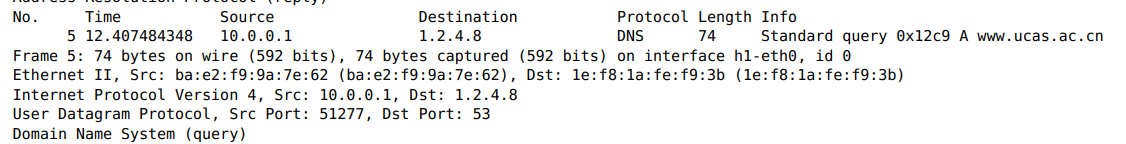


Figure 查询

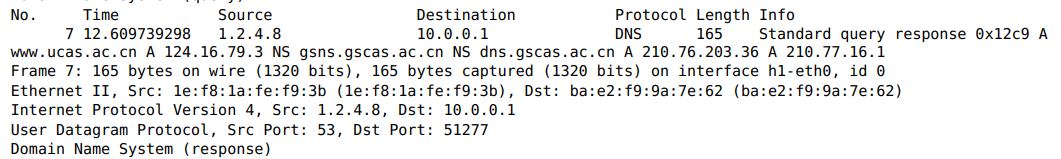


Figure 返回目标ip 124.16.79.3

若本地服务器没能找到记录，则将请求转发至根服务器，接下来是顶级服务器，权限服务器。递归过程的最后会把ip返回给客户机。客户机不知道查询的过程。至此h1获得了[www.ucas.ac.cn的ip](http://www.ucas.ac.cn的ip)地址，接下来可以与地址对应的服务器建立联系。

* TCP协议是传输层的协议，通过TCP协议建立的虚通道，两个端点的传输层可以建立联系。TCP报文构成如下：

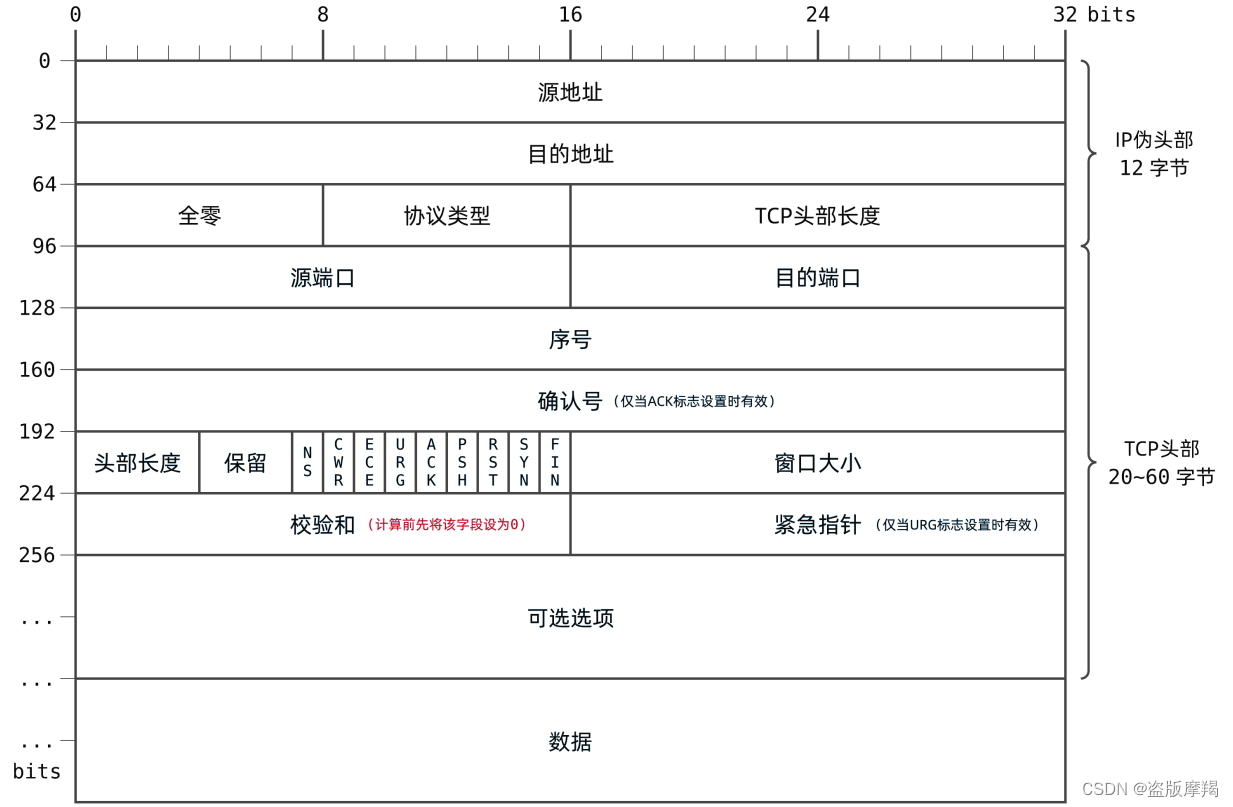


Figure TCP报文

为了与服务器建立联系，TCP协议要求客户机与服务器之间进行“三次握手”。第一次握手由客户发送给服务器，将上述报文中的序号（seq）设为一个随机数A，并将SYN标志置起，表示发起同步请求。

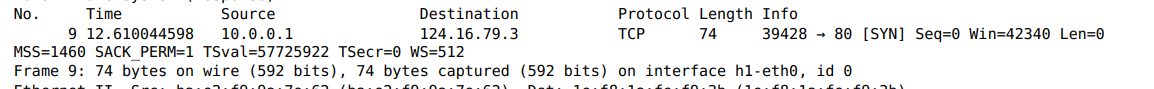


Figure seq=0，SYN=1

第二次握手是服务器回应客户机请求，将SYN和ACK标志置起，确认号（ack）置为A+1，seq设为随机数B，发送给客户机。这代表服务器确认建立连接。

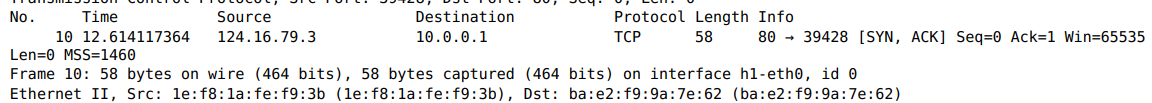


Figure seq=0，ack=1，SYN=ACK=1

第三次握手再由客户机发送给服务器，再次确认连接成功。客户机检查ack==A+1以及ACK==1后，将ack置为B+1并置起ACK，发送给服务器。服务器检查ack==B+1后，连接建立完成。

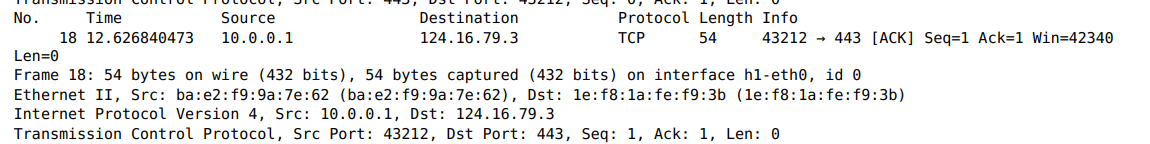


Figure 连接已建立

当建立连接后，可以在TCP协议的支持下进行数据传递。多数是由服务器向客户机发包，然后客户机发包确认收到。数个循环之后传输完毕。其中一个收发如下：

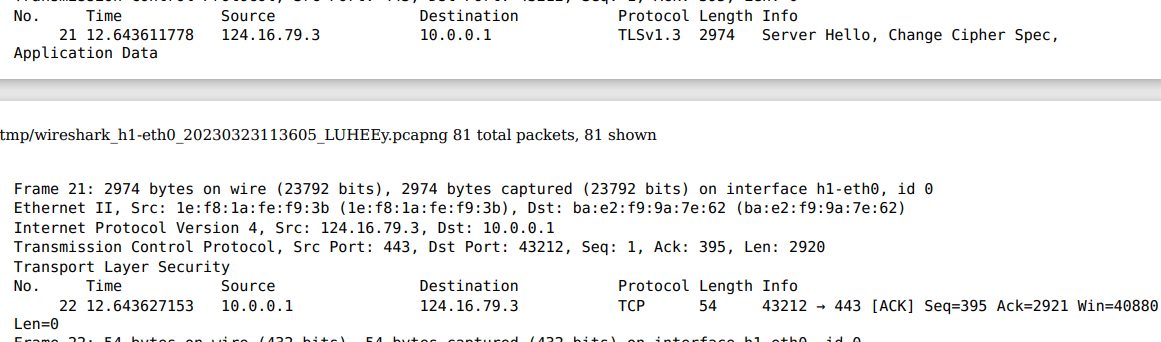


Figure 服务器发送数据长度为2920字节，客户回答ack=2921表示前2921个字节收到

当传输结束时，TCP协议要求服务器和客户机之间进行“四次挥手”以终止连接。过程见下图：

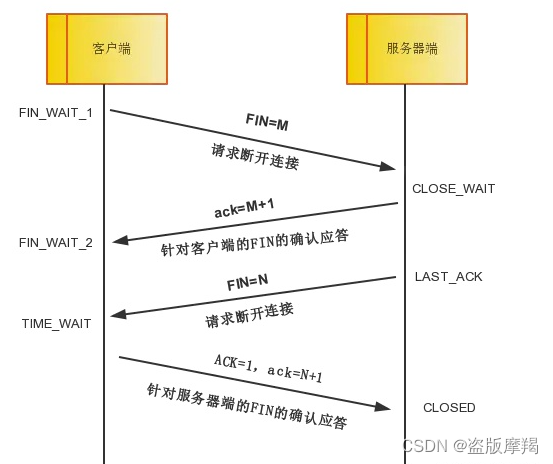


Figure 终止连接

其中如果第二次挥手后服务器没有信息要再送给客户，可以把第二、第三次挥手合并为一个包。

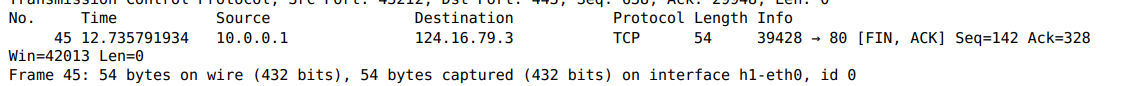


Figure 挥手1

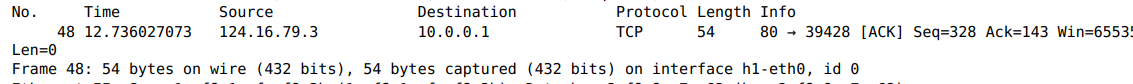


Figure 挥手2

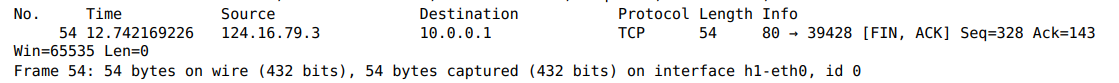


Figure 挥手3

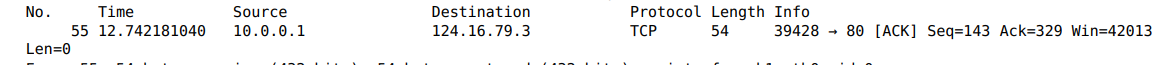


Figure 挥手4

至此数据传输完毕，连接已关闭。wget过程基本结束。

* 在TCP握手建立连接之前，客户机应先发送HTTP请求给服务器，建立应用层连接。本例中，发送的 HTTP请求是GET，要求服务器发送页面内容给客户机。此外，HTTP1.1还支持POST, PUT, HEAD, DELETE, OPTIONS, TRACE和CONNECT方法。GET方法会收到服务器的应答，其中包含一个状态码，2开头代表正常，其余均为异常。在本例中，HTTP请求收到了异常码302：

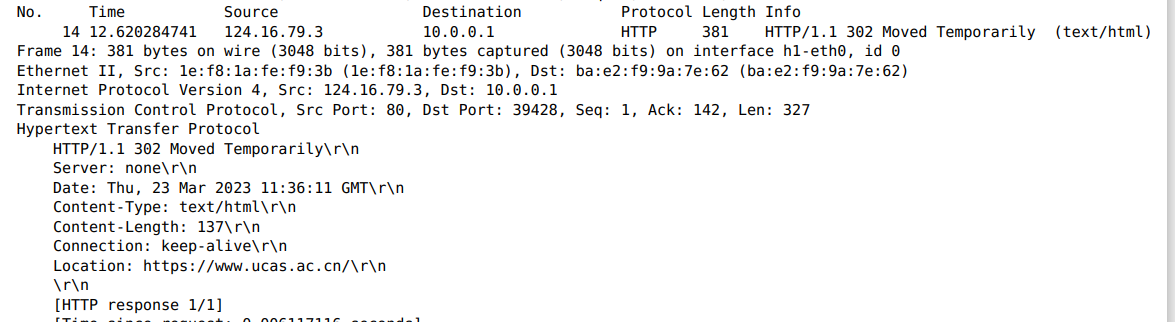


Figure 302

它表示页面内容被移动了，访问可能存在问题。进一步探索知道，wget为域名附加了http头部，它对应的默认端口是80，而ucas主页只接受https默认端口443的访问。服务器把这个改进建议随上述HTTP应答一起送回来，然后h1根据这个建议把协议改成了https，这次成功了：

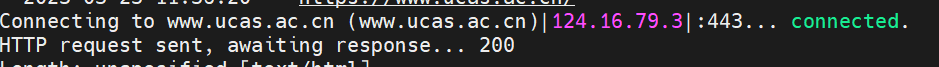


Figure 端口443回应状态码200

至此，这个wget过程基本分析完毕了。得到的主页数据被存放在执行mn的目录下。

1. **流完成时间实验**
2. 实验操作

使用sudo pip3 install mininet安装脚本需求，之后执行该脚本，在mininet中创建两个虚拟主机h1与h2，其中h2的ip为10.0.0.2。使用dd命令创建数据文件后，在h1中使用wget命令从h2下载该数据文件，并记录传输用时。之后，调整文件大小和带宽限制参数以改变自变量。实验全过程基本都是重复的，下面为实验中某一刻的截图：

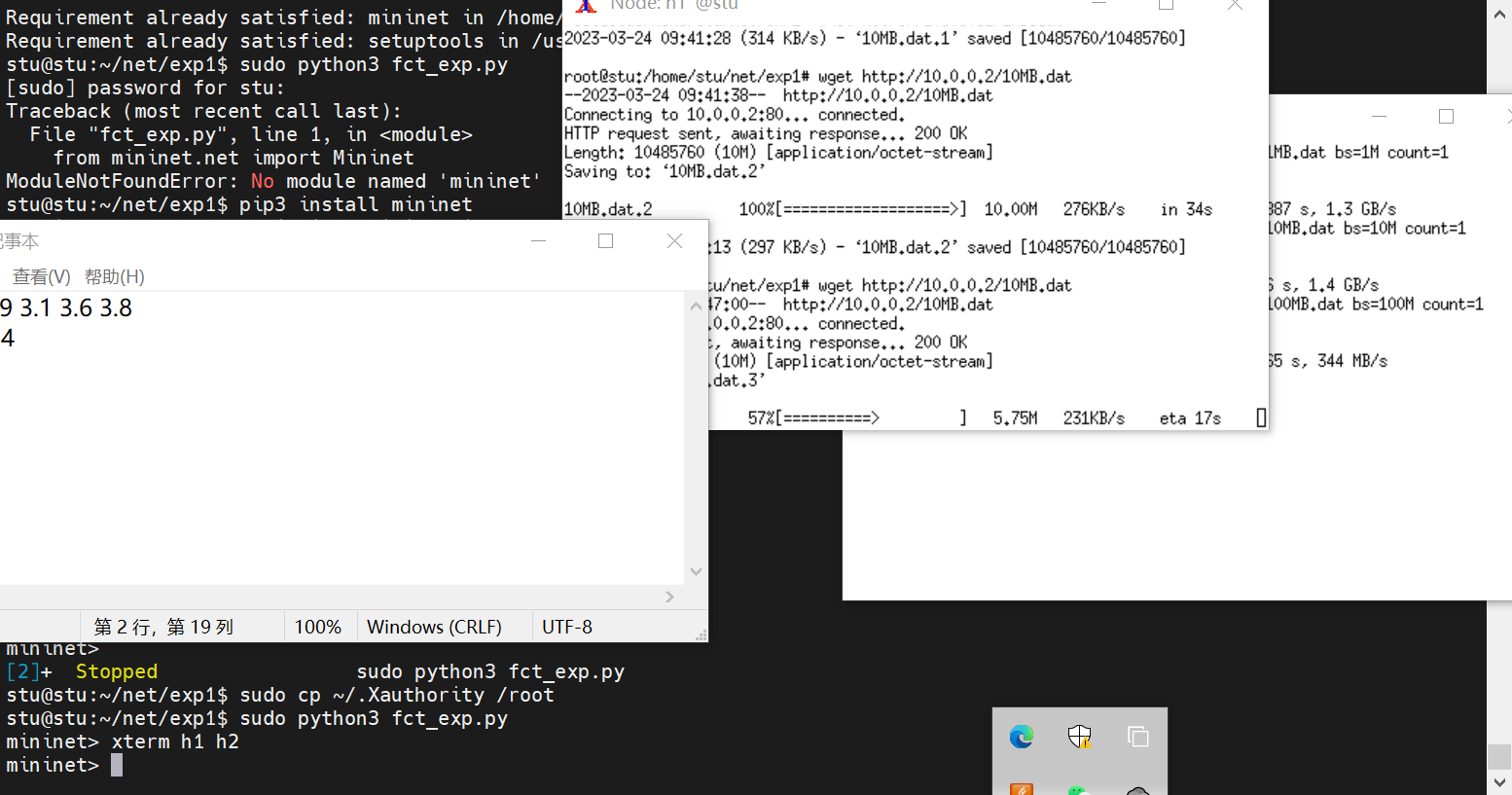


Figure 实验刚开始时的一个瞬间

1. 复现图像

待复现的图像如下：

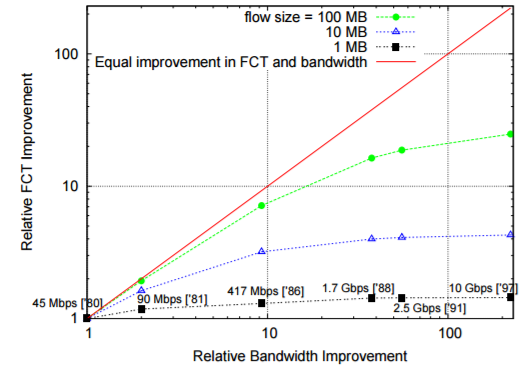


Figure 论文中的图像

其中横坐标为带宽的提升，纵坐标为流完成时间的提升，均为log坐标。

根据步骤1获得的数据，我使用python的matplotlib绘制了图像，同样采取对数坐标轴，效果如下：

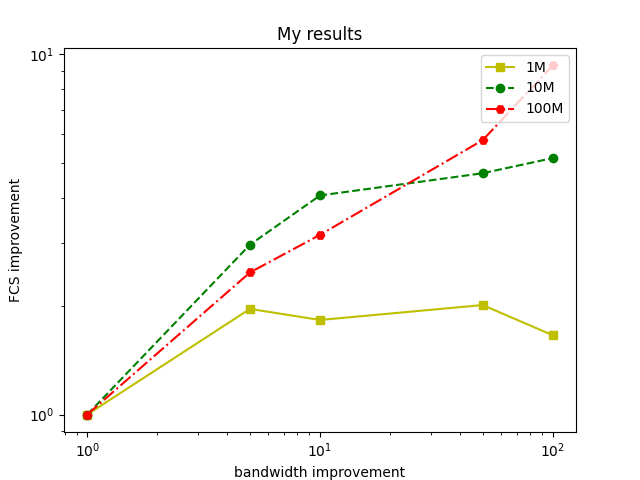


Figure 复现效果

附代码如下：

#code

import matplotlib.pyplot as plt  
x = [1, 50/10, 100/10, 500/10, 1000/10]  
y1 = [1, (1/1.64)/(1/3.22), (1/1.76)/(1/3.22), (1/1.6)/(1/3.22), (1/1.94)/(1/3.22)]  
y10 = [1, (1/12)/(1/35.5), (1/8.76)/(1/35.5), (1/7.6)/(1/35.5), (1/6.9)/(1/35.5)]  
y100 = [1, (1/90.8)/(1/225), (1/71.4)/(1/225), (1/39)/(1/225), (1/24.2)/(1/225)]fig = plt.figure()ax = fig.add\_axes([0.1,0.1,0.8,0.8])ax.set\_xscale("log")  
ax.set\_yscale("log")  
l1 = ax.plot(x,y1,'ys-')  
l2 = ax.plot(x,y10,'go--')  
l3 = ax.plot(x, y100, 'rH-.')  
ax.legend(labels = ('1M', '10M', '100M'), loc = 'upper right')ax.set\_title("My results")  
ax.set\_xlabel("bandwidth improvement")  
ax.set\_ylabel("FCS improvement")  
plt.show()

尽管由于本地条件限制以及实验次数比较少，复现的效果并不是很理想，但复现图和原图基本上都反映了同样的问题，即：

1. 实际吞吐率的提升低于带宽的提升；
2. 传输的总数据量越大，吞吐率提升越接近理想情况。
3. 分析

wget数据传输使用TCP协议，而吞吐率不如带宽是TCP传输中常见的现象，其原因之一在于TCP的慢启动机制对发送方做出的限制。网络的负载能力是有限的，如果所有数据都不加限制地放到网络上，容易出现拥塞，最终导致死锁。因此，需要用拥塞控制算法管理各个节点以及网络本身，尽可能避免拥塞情况的产生。拥塞控制算法是一个复杂的算法，其中一个部分就是对发送方的慢启动限制。

发送方维护一个拥塞窗口，发送方的发送窗口不得大于拥塞窗口。慢启动算法（RFC 5681标准）规定，拥塞窗口的初始值不超过2至4个发送方的最大报文段SMSS。之后每当收到确认时，可以扩大拥塞窗口，但每次扩大的规模不会超过一个SMSS。应该指出的是，在慢开始阶段，拥塞窗口的增长是指数级的。简单来说，就是发送方先以小规模发送数据，以探查网络的拥塞程度，之后扩大发送规模直到感受到拥塞影响为止。

这就很好地解释了步骤2中的现象：由于拥塞算法控制，无论带宽多大，发送方一开始总是以一个很小的发送速率启动，造成吞吐率较小。对于数据总规模本身就很小的传输，考虑指数图像的积分，往往在增长幅度还很小的情况下就结束了传输，因此带宽的变化基本上对传输时间没什么影响；而对于数据规模相对较大的情况，后期传输的限制因素可能为带宽，带宽增大就可以比较明显地影响到传输的时间。

最后，在拥塞窗口较大，而与预设的慢开始门限ssthresh相等时，拥塞算法将启用拥塞避免机制，在此阶段拥塞窗口只能线性增长。因此可以预见的是，即使数据规模很大，吞吐率随带宽增大的增长效果也很难达到理想情况。