基于套接字的网络程序设计

方桂安*，古博老师†

中山大学 智能科学与技术 20354027

【摘　要】 本次实验中我编写了一个基于 UDP 的 ping 程序，并计算出最大、最小、平均的往返时延 RTT 和丢包率。以及一个基于 TCP 的 Web 服务器程序，并使其在相同时间内能够处理多个请求。通过两个程序的编写，我在实践中加深了对套接字基本概念的理解，并且了解了基于 python 的套接字使用方法。

【关键词】 **套接字**，UDP，TCP，**通信**，Web **服务器**

1　引 言

socket[1] 是一套用于不同主机间通信的 API，它工作在我们的 TCP/IP 协议栈之上，它的应用无处不在。比如浏览器，手机应用，用于服务器管理的 SSH 客户端，全都是基于 socket 实现的。通过 socket 我们可以建立一条用于不同主机，不同应用之间的虚拟数据通道，并且是点对点（应用对应用）的。我们经常用到的 socket 有两种：TCP 和 UDP。接下来我将通过 TCP 和 UDP 的实例来学习它们的用法。

2　实验：套接字基础与 UDP 通信

本实验附件中展示了一段 python 代码，实现了一个 UDP 服务器，该服务器还会模拟丢失 30% 的客户端数据包。我们需要据此编写一个客户端程序，使用 UDO 发送 ping 消息，并计算出最大、最小、平均的 RTT 和丢包率。

2.1　代码与说明

首先，我在 python 的官方文档中了解了 socket 的相关函数的使用方法，然后根据自己的理解和编程习惯重新编写了 UDP 服务端，以下是我的代码：

图 1　socket 的官方文档

```python
# 创建一个 UDP 套接字 （SOCK_DGRAM）
with socket(AF_INET, SOCK_DGRAM) as s:
    # 绑定端口
    s.bind(('0.0.0.0', 12000))
    while True:
        rand = random.randint(0, 10)
        # 接收 UDP 数据，其中 message 是包含接收数据的字符串，address 是发送数据的套接字地址。
        message, address = s.recvfrom(1024)
        # 打印message
        print("Received: %s" % message.decode("utf-8"))
        # 字符串中的小写字母转为大写字母。
        message = message.upper()
        # 模拟 30% 的数据包丢失
        if rand < 4:
            continue
        s.sendto(message, address)
```

使用 with 语法糖，我们可以自动关闭套接字。AF_INET 表示使用 IPv4 协议，SOCK_DGRAM 表示使用 UDP 协议。bind() 函数绑定套接字到指定的地址和端口。值得注意的是'0.0.0.0' 表示绑定到

实验时间: 2022-04-12
报告时间: 2022-04-26
† **指导教师**
***学号**：20354027
*E-mail：fanggan@mail2.sysu.edu.cn

本机所有的 IP 地址上。

```
# 创建一个 UDP 套接字 （SOCK_DGRAM）
with socket(AF_INET, SOCK_DGRAM) as c:
    # 使用settimeout函数限制recvfrom()函数的等待时间为1秒
    c.settimeout(1)
    # 记录RTT
    rttTimes = []
    # 记录丢包次数
    count = 0
    # 发送10次ping报文
    for i in  range(10):
        # RRT开始计数时间
        sendTime = time.perf_counter()
        # 将信息转换为byte后发送到指定服务器端
        c.sendto("ping".encode("utf-8"),("127.0.0.1",12000))
        try:
            # 调用recvfrom()函数接收服务器发来的应答数据
            message,address=c.recvfrom(1024)
            # 打印message
            print("Received: %s" % message.decode("utf-8"))
        #超时处理，等到时间超过1秒，捕获抛出的异常后打印丢失报文，进行下一步操作
        except:
            print("请求超时!")
            count += 1
            continue
        # 计算往返时间
        rtt = time.perf_counter() - sendTime
        rttTimes.append(rtt)
        print('RTT = %.15f'%rtt)
# 计算ping消息的最小、最大和平均RRT，并计算丢包率
mintime =  min(rttTimes)
maxtime =  max(rttTimes)
avetime =  sum(rttTimes) /  len(rttTimes)
print('min_RRT:'+ str(mintime))
print('max_RRT:'+ str(maxtime))
print('average_RRT:'+ str(avetime))
print('packet loss rate:'+ str(count)+'0%')
```

服务端会将接收到的数据包打印在终端，然后转换为大写字母，并返回给客户端。rand 是一个范围在 0 到 10 的随机整数，如果小于 4，则模拟数据包丢失，跳出循环而不返还数据。

接下来是我编写的客户端代码，相关的思路写在了注释中。开发过程中我使用的是同一台电脑进行测试，故 ip 绑定在 127.0.0.1 上。

客户端会向服务端发送 10 个 “ping” 报文，服务端如果在 1 秒内响应，则打印该响应消息，否则打印 “请求超时”。最后客户端会再打印收到的 “PING” 报文，并计算、打印这一过程的 RTT。

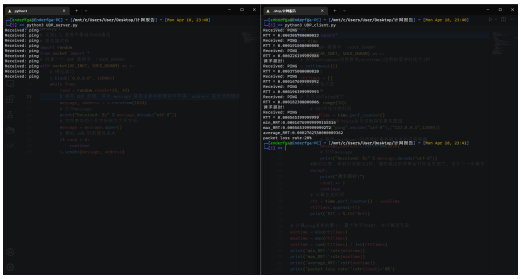

图 2 windows

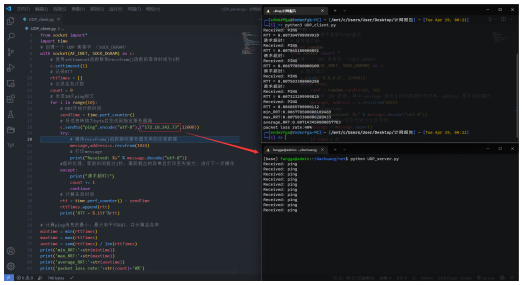

图 3 linux

其中我使用了 perf_counter() 函数来计算 RTT，它返回当前时间，单位是秒。通过列表 rttTimes 来记录所有数据，以便于最大、最小和平均 RTT 的计算。

2.2 结果与分析

首先我将写好的代码在 windows 环境下同时运行，结果如图2所示。左侧的服务端收到并打印出了 10 个 “ping” 报文，右侧的客户端输出了 8 个 “PING” 报文和对应的往返时延，以及 2 次请求超时，所得数据如表1：

表 1 windows:RTT 及丢包率

min_RTT	max_RTT	avg_RTT	packet loss rate
0.000168	0.000565	0.000276	20%

然后我将客户端改为实验室中的 linux 环境下运行，结果如图3所示。显然 RTT 的各项数据都比 windows 环境下的数据大一个数量级，具体数据如表2：

接下来我又将服务端，客户端运行环境对调，并在 linux 环境下同时运行两份代码。总共四组实验，并多次重复试验确保数据具有普遍性。最终得到以下结论：

表 2 linux:RTT 及丢包率

min_RTT	max_RTT	avg_RTT	packet loss rate
0.006779	0.007503	0.007143	40%

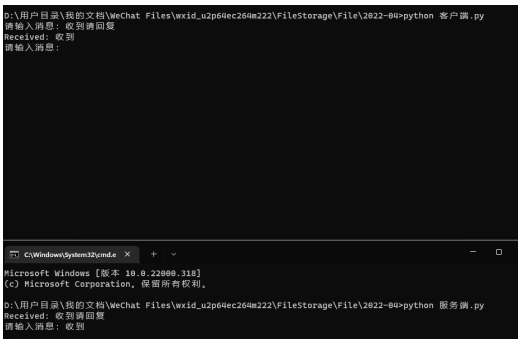

图 4 UDPchat

- UDP 延迟小于 1ms，但网络阻塞依旧存在，跨主机通信时更明显。
- 本次实验中丢包是由随机数产生的，故可能出现 0% 甚至 100% 的情况。

2.3 问题与改进

附件的服务端代码将收到的 message 转换成大写字母，并将其发送回客户端。无法在终端上清晰地展现结果，故我让程序在发送前将收到的报文打印，再让客户端打印收到的报文，从而实现通信效果。只要将默认的“ping”报文修改为 input() 函数，即可成为一个简易的“聊天软件”。为了用户使用体验，还可以加入心跳机制，效果如图4。

往返时延是指客户发送 ping 报文到接受到 pong 报文为止的时间，因此我记录了这一过程的起止时间来计算 RTT，并通过 min(),max() 等函数来计算最值。而一开始我使用的是 time.time(), 该函数返回自纪元以来的秒数作为浮点数，但是时期的具体日期和闰秒的处理取决于使用的平台。比如：在 Windows 和大多数 Unix 系统上，纪元是 1970 年 1 月 1 日 00:00:00（UTC），并且闰秒不计入自纪元以来的秒数，这也通常被称为 Unix 时间。可即使时间总是作为浮点数返回，但并非所有系统都提供的精度高于 1 秒，而且更改系统的时间会影响 time() 的值。

对此我提出了一个改进，将 time.time() 改为 time.per_counter(), 该函数返回性能计数器的值（以小数秒为单位）作为浮点数，即具有最高可用分辨率的时钟，以测量短持续时间。

3 实验：TCP 通信与 Web 服务器

本实验附件中展示了一份不完整的 Web 服务器框架代码。我们需要逐步填充代码中不完善的部分，并完成一个简单的 Web 服务器。最终实现能够同时处理多个请求的多线程服务器。

3.1 代码与说明

SOCK_STREAM 提供面向连接的稳定数据传输，即 TCP 协议。与 UDP 不同，TCP 需要建立连接后才能发送或接收数据，故在 bind() 后 accept() 前还添加了 listen() 函数，用于监听客户端的连接请求。

```
1    # 创建一个 TCP 套接字 （SOCK_STREAM）
2    with socket(AF_INET, SOCK_STREAM) as s:
3        # 将TCP欢迎套接字绑定到指定端口
4        s.bind(('0.0.0.0', 12000))
5        # 设置最大连接数为5
6        s.listen(5)
7        while True:
8            # 建立连接
9            print('Ready to serve...')
10           # 接收到客户连接请求后，建立新的TCP
                 连接套接字
11           connectionSocket, addr = s.accept()
12           # 创建新线程处理客户端请求,以实现多
                 线程服务器
13           t = threading.Thread(target=
                 handle_client, args=(
                 connectionSocket,addr))
14           t.start()
```

socket 的 accept 函数每次只能接收一个 socket，如果同一时间有大量连接进入，socket 需要使用缓存队列的方式（线程池），将还未及时处理的连接保存在队列中，而队列的大小就是 listen 参数的值了，大部分应用程序默认设为 5。

我们进行程序开发的时候，肯定避免不了要处理并发的情况。一般并发的手段有采用多进程和多线程。但线程比进程更轻量化，系统开销一般也更低，所以大家更倾向于用多线程的方式处理并发的情况。Python 提供多线程编程的方式。需要借助于 threading 模块。

我们要创建 Thread 对象，然后让它们运行，每个 Thread 对象代表一个线程，在每个线程中我们可以让程序处理不同的任务，这就是多线程编程。

图 5 HTTP 报文

值得注意的是，程序运行时默认就是在主线程上。创建 Thread 时需将一个 callable 对象从类的构造器传递进去，这个 callable 就是回调函数，用来处理任务。对于 http 请求的处理我编写在回调函数 handle_client() 中。

```
1  def handle_client(client_socket,addr):
2      with client_socket:
3          # 获取客户发送的报文
4          message = client_socket.recv(1024)
5          # 获取客户端发送的请求行
6          filename = message.split(b"\r\n")
7          outputdata = filename[0].split()[1].
               decode()
8          # 映射“根”目录默认html文件
9          if outputdata == '/':
10             outputdata = 'index.html'
11         else :
12             outputdata = outputdata[1:]
13         try:
14             with  open(outputdata,'rb') as f:
15                 # 读取文件内容
16                 content = f.read()
17                 # 发送响应行
18                 response = b'HTTP/1.1 200 OK\r\
                       n' + b'Content-Type: text/
                       html\r\n' + b'\r\n' +
                       content
19         except FileNotFoundError:
20             with  open('404.html','rb') as f:
21                 # 读取文件内容
22                 content = f.read()
23             # 发送未找到文件的响应消息
24             response = b'HTTP/1.1 404 Not Found
                   \r\n'+ b'\r\n' + content
25         client_socket.sendall(response)
```

如图5是浏览器客户端发送的报文格式，值得注意的是报文是通过回车符和换行符分割的，即\r\n。所以我使用\r\n作为报文的分隔符，这样就分割报文并解析出客户端请求的文件名。

客户端发来的 HTTP 请求中，url 都是相对根“/”的相对路径，如果用户请求的是根路径，我们则直接返回 index.html 文件，如果用户请求的是其他路径，我们则返回对应的文件。

图 6 三次请求的结果

响应报文的格式与请求报文相同，通过在 python 普通字符串之前添加前缀 b，我们将其数据类型从字符串修改为字节。

b 字符串和 Python 字符串有一个主要区别，那就是它的数据类型。普通字符串具有 Unicode 字符序列，而 Python b 字符串具有字节数据类型。两者可以通过 encode() 和 decode() 转换。

然后我们根据不同的结果返回对应的状态码，假如是 404，我们则返回 404.html 文件。最后 sendall 函数会将“组装”好的全部信息发送给客户端。

3.2 结果与分析

本次实验中我编写了三个 html 文件，分别是 index.html、404.html 和指定的 hw.html，首先是模拟用户直接访问 http://ip:port，默认页面会显示一句随机诗词；然后是指定文件，即 http://ip:port/hw.html，页面显示“Hello World!”；最后是 404 页面，即 http://ip:port/rand.html（任意指定一个不存在的文件），页面显示“404 Not Found!”。三次请求的结果如图6。

为了验证服务器是否能完成多线程的任务，我同时打开了 4 个浏览器客户端，同时访问 Web 服务器的主页，结果如图7。

虽然服务端的终端此时出现了四份连接日志，但由于我不能确保多个客户端是否同时发出请求，因此我编写了一个简易的浏览器客户端，发出请求之后会保存对应的 html 文件，以实现服务器的

图 7 多线程访问结果

```
Ready to serve...

Accept new connection from 127.0.0.1:3642...
Ready to serve...

Accept new connection from 127.0.0.1:3643...
Ready to serve...

Accept new connection from 127.0.0.1:3645...
Ready to serve...

Accept new connection from 127.0.0.1:3646...
Ready to serve...
```

图 8 服务端终端输出

并发处理。

```
with socket(AF_INET, SOCK_STREAM) as s:
    # 将TCP欢迎套接字绑定到指定端口
    url =  input('请输入要访问的网址：')
    IP = url.split('/')[2].split(':')[0]
    PORT =
        int(url.split('/')[2].split(':')[1])
    s.connect((IP, PORT))
    # 指定要访问的文件
    filename = url.split('/')[3]
    # 组装请求报文
    request = ('GET /'+filename+' HTTP/1.1\r\n'
    +'Host: '+IP+':'+ str(PORT)+'\r\n\r\n')
    # 发送请求行
    s.send(request.encode('utf-8'))
    # 接收响应行
    response = s.recv(1024)
    # 如果文件存在，则接收文件内容，否则输出错
        误信息
    if response.decode('utf-8').startswith('
        HTTP/1.1 200 OK'):
        with open(filename, 'wb') as f:
                content = str(response.decode('
                    utf-8').split('\r\n\r\n')
                    [1])
                f.write(content.encode('utf-8')
                    )
    if response.decode('utf-8').startswith('
        HTTP/1.1 404 Not Found'):
        print('File not found')
```

该客户端程序需要输入完整的网址，如 http://ip:port/hw.html，若文件不存在，终端会输出“File not found”。反之，则将文件保存到客户端程序所在目录下。

3.3 问题与改进

在本地测试多次之后，我开始从其他设备来访问我的 web 服务器，然而出现了访问失败的情况，对此，我也将 html 文件及服务端程序转移到 linux 服务器中运行，由于服务器具有公网 IP，无论通过电脑、手机还是平板等设备，都可以轻松访问到目标 html 页面，借此我成功搭建了一个用于个人博客的简易 web 服务器。

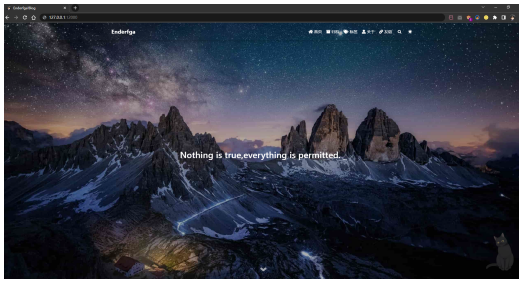

图 9 Blog

另外，通过查阅资料我得知由于 GIL(Global Interpreter Lock，全局解释器锁) 的存在，python 中的线程其实做不到真正的并发，并且线程自身也会占用额外的系统资源，因为每个线程的绝大多数时间都是在等待这些 IO 操作，且线程的切换也会有额外的开销，更不用说线程之间的资源竞争问题。而 Web 应用需要经常执行网络操作、数据库访问，故可以考虑使用异步编程的方式。

此外我还发现 python 其实自带了一个 http 模块，这个框架主要由 server 和 handler 组成。server 主要用于建立网络模型，例如利用 epoll 监听 socket;handler 用于处理各个就绪的 socket。

4 结 论

本次实验我编写了一个 UDP_ping 程序和一个 TCP_web 程序，深刻体会了 socket 传输的实时性、高效性与安全性。通过套接字的基础编程，更深地了解了自顶向下的计算机网络思想。

参考文献

[1] KUROSE J, ROSS K. 计算机网络: 自顶向下方法 [M]. [出版地不详]: 计算机网络: 自顶向下方法, 2009.