



警示

- 1.实验报告如有雷同，雷同各方当次实验成绩均以 0 分计。
- 2.当次小组成员成绩只计学号、姓名登录在下表中的。
- 3.在规定时间内未上交实验报告的，不得以其他方式补交，当次成绩按 0 分计。
- 4.实验报告文件以 PDF 格式提交。

院系	数据科学与计算机学院	班 级	16 级信息与计算科学	组长	回煜淼
学号	16339021	16339049	16343065		
学生	回煜淼	辛依繁	桑娜		

编程实验

【实验名称】

基于UDP丢包统计程序设计

【实验目的】

选择一个操作系统环境（Linux或Windows），编制UDP/IP通信程序，完成一定的通信功能。

【实验要求】

在发送UDP数据包时做一个循环，连续发送100个数据包；在接收端统计丢失的数据包。

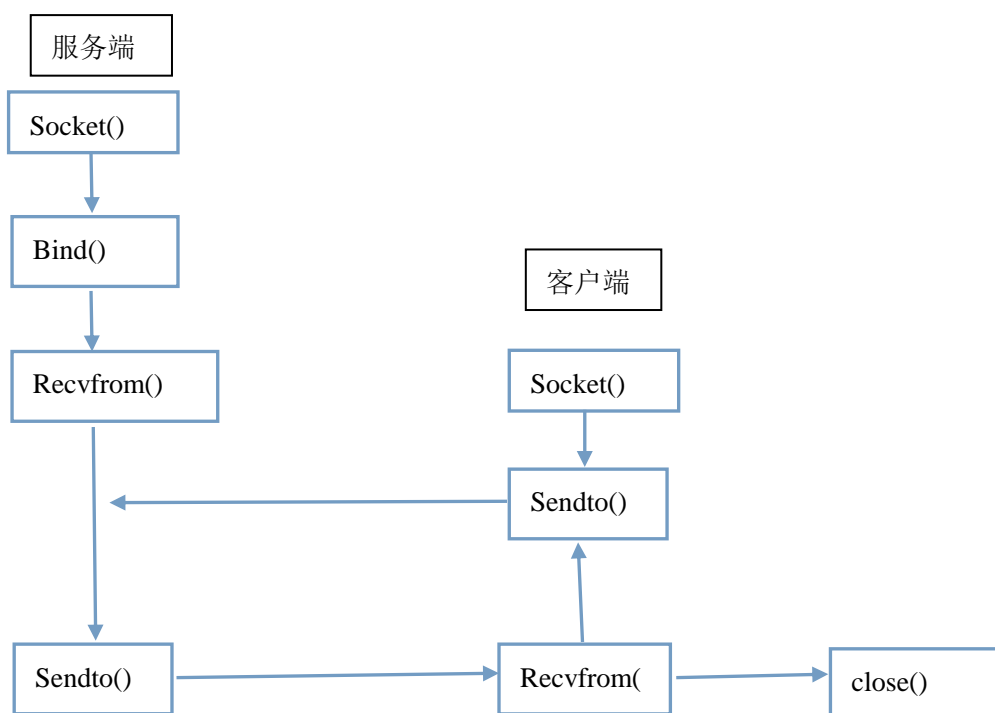
实验时，请运行Wireshark软件，对通信时的数据包进行跟踪分析。

【实验思路】

参考教程与示例代码，利用socket进行编程

服务端：建立套接字——绑定ip地址和端口——等待——接收数据——应答——结束

客户端：建立套接字——发送数据——接收应答——结束





【服务端】

服务端接收客户端传来的数据并回复客户端，代码如下：

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <winsock2.h>
4  #define BUFFER_SIZE 1024
5  #pragma comment(lib,"ws2_32")
6  using namespace std;
7  int main(){
8      // socket初始化检查
9      WSADATA wsaData;
10     if(WSAStartup(MAKEWORD(2,2),&wsaData)){ // 初始化失败
11         cout<<"socket初始化失败"<<endl;
12         return 0;
13     }
14
15     // 1. 创建套接字
16     SOCKET ser_socket = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
17     if(ser_socket == INVALID_SOCKET){
18         cout << "INVALID_SOCKET" << endl;
19         return 0;
20     }
21
22     // 2. 将本地IP地址和端口号绑定到套接字
23     sockaddr_in ser_addr = {AF_INET, htons(6000)};
24     ser_addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
25     if(bind(ser_socket,(sockaddr*)&ser_addr,sizeof(ser_addr))== SOCKET_ERROR){
26         cout << "SOCKET_ERROR"<<endl;
27         closesocket(ser_socket);
28         return 0;
29     }
30
31     sockaddr_in clt_addr; // 用来获取客户端地址
32     char Buf[BUFFER_SIZE]; // 接收数据缓冲区
33     int i = 0; // 数据包计数器
34     const char respond[] = "数据已收到"; // 应答消息
35     int addr_size = sizeof(sockaddr);
36
37     // 3. 等待客户数据
38     while(1) {
39         // 处理请求
40         int len = recvfrom(ser_socket,Buf,BUFFER_SIZE,0,(sockaddr*)&clt_addr,&addr_size);
41         if(len > 0){
42             Buf[len] = '\0';
43             cout << "接收到来自"<<inet_ntoa(clt_addr.sin_addr)<<"的数据: "<<Buf<<endl;
44             cout <<"已收到"<< ++i << "个数据包" <<endl;
45             // 应答
46             sendto(ser_socket,respond,strlen(respond),0,(sockaddr*)&clt_addr,addr_size);
47         }
48     }
49     closesocket(ser_socket);
50     WSACleanup();
51     return 0;
52 }
```



【客户端】客户端主要是确定好服务端地址，然后传递数据；如果服务器接收成功那么给予客户端响应，代码如下所示：

```
1  #include <winsock2.h>
2  #include <stdio.h>
3  #include <iostream>
4  #pragma comment(lib, "ws2_32.lib")
5  #define BUFFER_SIZE 1024
6  using namespace std;
7  int main (){
8      // socket初始化检查
9      WSADATA wsaData;
10     if(WSAStartup(MAKEWORD(2,2), &wsaData) != 0){
11         cout << "初始化失败" << endl;
12         return 0;
13     }
14
15     // 1. 建立套接字
16     SOCKET clt_sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
17     if(clt_sock == INVALID_SOCKET){
18         cout << "INVALID_SOCKET" << endl;
19         return 0;
20     }
21
22     // 设定服务器地址
23     sockaddr_in ser_addr = {AF_INET, htons(6000)};
24     ser_addr.sin_addr.S_un.S_addr = inet_addr("172.18.138.9");
25
26     sockaddr_in recvFrom_addr; // 获取应答消息来源地址
27     int addr_size = sizeof(sockaddr);
28
29     // 2. 发送数据
30     int i = 0;
31     while(i < 100){
32         char sendBuf[BUFFER_SIZE]; // 发送缓冲区
33         itoa(i, sendBuf, 10);
34         sendto(clt_sock, sendBuf, strlen(sendBuf), 0, (sockaddr*)&ser_addr, addr_size);
35         cout << "发送数据给服务端: " << sendBuf << endl;
36         // 接收应答
37         char recvBuf[BUFFER_SIZE];
38         int len = recvfrom(clt_sock, recvBuf, BUFFER_SIZE, 0, (sockaddr*)&recvFrom_addr, &addr_size);
39         if(len > 0){
40             recvBuf[len] = '\0';
41             cout << "接收到来自" << inet_ntoa(recvFrom_addr.sin_addr) << "的应答: " << recvBuf << endl;
42         }
43         i++;
44     }
45     closesocket(clt_sock);
46     WSACleanup();
47     return 0;
48 }
```

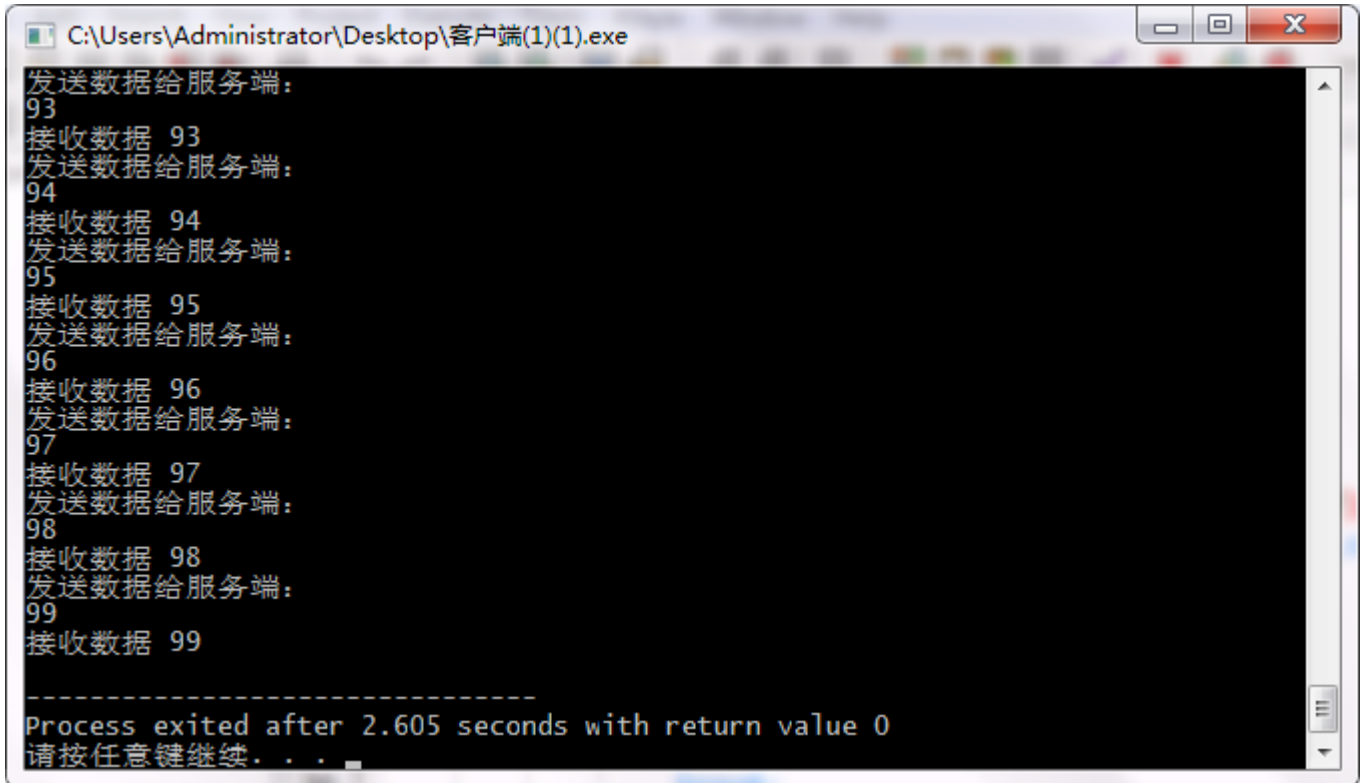
(注：此处代码为最终整理版本，故与下面的实验结果及分析截图有些许出入)



【实验结果及分析】

【A-局域网实验】

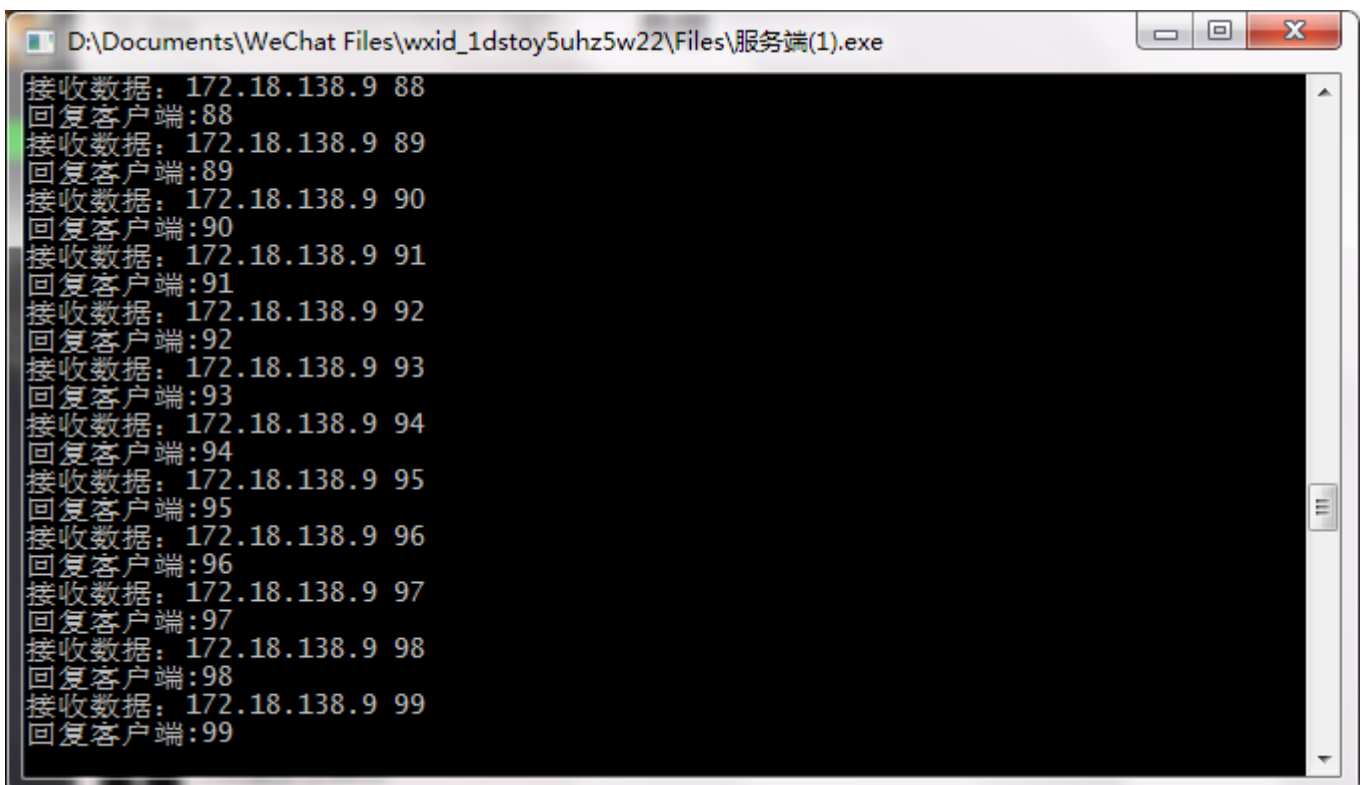
A.1 客户端内网IP: 172.18.138.9, 端口为1000. 向服务端发送100个连续的数据包, 发送成功得到服务端回复。



```
C:\Users\Administrator\Desktop\客户端(1)(1).exe
发送数据给服务端:
93
接收数据 93
发送数据给服务端:
94
接收数据 94
发送数据给服务端:
95
接收数据 95
发送数据给服务端:
96
接收数据 96
发送数据给服务端:
97
接收数据 97
发送数据给服务端:
98
接收数据 98
发送数据给服务端:
99
接收数据 99

-----
Process exited after 2.605 seconds with return value 0
请按任意键继续. . .
```

A.2 服务端内网IP: 172.18.154.95, 接受到了服务端发送的100个数据包并回复客户端。



```
D:\Documents\WeChat Files\wxid_1dstoy5uhz5w22\Files\服务端(1).exe
接收数据: 172.18.138.9 88
回复客户端:88
接收数据: 172.18.138.9 89
回复客户端:89
接收数据: 172.18.138.9 90
回复客户端:90
接收数据: 172.18.138.9 91
回复客户端:91
接收数据: 172.18.138.9 92
回复客户端:92
接收数据: 172.18.138.9 93
回复客户端:93
接收数据: 172.18.138.9 94
回复客户端:94
接收数据: 172.18.138.9 95
回复客户端:95
接收数据: 172.18.138.9 96
回复客户端:96
接收数据: 172.18.138.9 97
回复客户端:97
接收数据: 172.18.138.9 98
回复客户端:98
接收数据: 172.18.138.9 99
回复客户端:99
```




计算机网络实验报告

运行 Wireshark，对通信时的数据包进行跟踪分析。发现客户端端口为：53986，服务端端口号为：1000。

视图(V)	跳转(G)	捕获(C)	分析(A)	统计(S)	电话(Y)	无线(W)	工具(T)	帮助(H)
18.138.9								
	Source	源地址	Destination	目的地址	Protocol	Length	Info	
0.487029	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.488929	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.499734	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.508241	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.513971	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.521197	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.533162	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.540914	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
0.551044	172.18.138.9		172.18.154.95		UDP	60	53986 → 1000 Len=2	
53: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0								
II, Src: Hangzhou_1b:a2:63 (0c:da:41:1b:a2:63), Dst: LcfcHefe_2a:57:c3 (68:f7:28:2a:57:c3)								
Protocol Version 4, Src: 172.18.138.9, Dst: 172.18.154.95								
gram Protocol, Src Port: 53986, Dst Port: 1000 服务端指定端口								
(te)								

A.3 结论：经统计，客户端发送的 100 个数据均收到，局域网内通信丢包率为 0。

【B-互联网实验】

B.1 客户端外网IP: 120.236.174.164，端口为5001. 向服务端发送100个连续的数据包，发送成功得到服务端回复。

```
C:\Users\Administrator\Desktop\客户端(1)(1).exe
发送数据给服务端:
93
接收数据 93
发送数据给服务端:
94
接收数据 94
发送数据给服务端:
95
接收数据 95
发送数据给服务端:
96
接收数据 96
发送数据给服务端:
97
接收数据 97
发送数据给服务端:
98
接收数据 98
发送数据给服务端:
99
接收数据 99
-----
Process exited after 0.4101 seconds with return value 0
请按任意键继续. . .
```

B.2 服务端外网IP: 119.23.204.50，接受到了服务端发送的100个数据包并回复客户端。



```
C:\Users\Administrator\Downloads\服务端.exe
回复客户端:88
接收数据:120.236.174.164 89
回复客户端:89
接收数据:120.236.174.164 90
回复客户端:90
接收数据:120.236.174.164 91
回复客户端:91
接收数据:120.236.174.164 92
回复客户端:92
接收数据:120.236.174.164 93
回复客户端:93
接收数据:120.236.174.164 94
回复客户端:94
接收数据:120.236.174.164 95
回复客户端:95
接收数据:120.236.174.164 96
回复客户端:96
接收数据:120.236.174.164 97
回复客户端:97
接收数据:120.236.174.164 98
回复客户端:98
接收数据:120.236.174.164 99
回复客户端:99
微软拼音 半 :
```

运行 Wireshark，对通信时的数据包进行跟踪分析。发现客户端端口为：36526，服务端端口号为：5001

Wireshark interface showing packet capture details for 120.236.174.164.

Time	Source	Destination	Protocol	Length	Info
46.37013.6613...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=5
51.37014.5780...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=4
52.37015.3428...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=3[Malf...
69.37016.3629...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=3[Malf...
70.37017.1594...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=5
71.37017.7087...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=3[Malf...
72.37017.9716...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=3[Malf...
73.37018.2139...	120.236.17...	172.18.200...	UDP	60	36526 → 5001 Len=2[Malf...

0146: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
t II, Src: ee:ff:ff:ff:ff:ff (ee:ff:ff:ff:ff:ff), Dst: Xensourc_0a:27:91 (00:16:3e:0a:27:
t Protocol Version 4, Src: 120.236.174.164, Dst: 172.18.200.42
tagram Protocol, Src Port: 36526, Dst Port: 5001
bytes)

B.3 进一步实验:



计算机网络实验报告

经统计互联网通信发送的 100 个数据服务端均收到，但由于数据量过小，所以为验证严密性，我们增加传输的数据量，我们利用一种软件 IPERF,其功能是用来进行 UDP 的监测。

B.4 参数说明：

我们设置了外网服务端地址为 119.23.204.50，采用 UDP 协议，带宽为 1M,端口为 6000，数据包为默认大小。

IPERF for Windows with Charts - Non-Commercial Trial

Controls | Console | Charts | About

-c 119.23.204.50 -u -b 1m -P 3 -t 19 -p 6000 -l 8k Start Terminate

☐ Server mode
☐ 0.0.0.0 bind to address

☒ Client mode
119.23.204.50 server or multicast address
1 MBits/sec target bandwidth
3 threads to run in parallel
1 time-to-live for multicast
10 time in sec to transmit for

☐ Help ☐ Version ☐ License

UDP protocol Defaults
6000 server port
8 KBytes datagram size
8 KBytes buffer size
2 DSCP value in decimal
IPv4 protocol version
MBytes format to report
2 seconds between reports
commands
// comments

Idle Exit

B.5 实验结果分析：



```
IPERF for Windows with Charts - Non-Commercial Trial
Controls Console Charts About
Clear Copy Export
UDP buffer size: 0.01 MByte (default)
-----本地局域网IP-----服务器IP
[320] local 172.18.154.95 port 62652 connected with 119.23.204.50 port 6000
[316] local 172.18.154.95 port 62651 connected with 119.23.204.50 port 6000
[ ID] Interval Transfer Bandwidth
[320] 0.0- 2.0 sec 0.24 MBytes 0.12 MBytes/sec
[316] 0.0- 2.0 sec 0.24 MBytes 0.12 MBytes/sec
[SUM] 0.0- 2.0 sec 0.48 MBytes 0.24 MBytes/sec
[320] 2.0- 4.0 sec 0.24 MBytes 0.12 MBytes/sec
[316] 2.0- 4.0 sec 0.24 MBytes 0.12 MBytes/sec
[SUM] 2.0- 4.0 sec 0.48 MBytes 0.24 MBytes/sec
[320] 4.0- 6.0 sec 0.23 MBytes 0.12 MBytes/sec
[316] 4.0- 6.0 sec 0.23 MBytes 0.12 MBytes/sec
[SUM] 4.0- 6.0 sec 0.47 MBytes 0.23 MBytes/sec
[320] 6.0- 8.0 sec 0.24 MBytes 0.12 MBytes/sec
[316] 6.0- 8.0 sec 0.24 MBytes 0.12 MBytes/sec
[SUM] 6.0- 8.0 sec 0.48 MBytes 0.24 MBytes/sec
[320] 8.0-10.0 sec 0.23 MBytes 0.12 MBytes/sec
[316] 8.0-10.0 sec 0.23 MBytes 0.12 MBytes/sec
[SUM] 8.0-10.0 sec 0.47 MBytes 0.23 MBytes/sec
[320] 10.0-12.0 sec 0.24 MBytes 0.12 MBytes/sec
[316] 10.0-12.0 sec 0.24 MBytes 0.12 MBytes/sec
[SUM] 10.0-12.0 sec 0.48 MBytes 0.24 MBytes/sec
[320] 12.0-14.0 sec 0.23 MBytes 0.12 MBytes/sec
[316] 12.0-14.0 sec 0.23 MBytes 0.12 MBytes/sec
[SUM] 12.0-14.0 sec 0.47 MBytes 0.23 MBytes/sec
[320] 14.0-16.0 sec 0.24 MBytes 0.12 MBytes/sec
[316] 14.0-16.0 sec 0.24 MBytes 0.12 MBytes/sec
[SUM] 14.0-16.0 sec 0.48 MBytes 0.24 MBytes/sec
[320] 16.0-18.0 sec 0.23 MBytes 0.12 MBytes/sec
[316] 16.0-18.0 sec 0.23 MBytes 0.12 MBytes/sec
[SUM] 16.0-18.0 sec 0.47 MBytes 0.23 MBytes/sec
[320] 0.0-19.1 sec 2.27 MBytes 0.12 MBytes/sec
[320] Sent 291 datagrams
[316] 0.0-19.1 sec 2.27 MBytes 0.12 MBytes/sec
[316] Sent 291 datagrams
[SUM] 0.0-19.1 sec 4.55 MBytes 0.24 MBytes/sec
[320] Server Report:
[320] 0.0-19.1 sec 2.20 MBytes 0.12 MBytes/sec 5.121 ms 10/ 291 (3.4%)
```

B.6 结论： IPERF 实时监听 UDP 协议数据的传输，最后得出结论：在本次实验条件下丢包率为 3.4%。

【实验思考】

一、说明在实验过程中遇到的问题和解决办法。

(1) 客户端和服务端程序无法通过编译

答：用Dev进行这次网络编程的时候，需要重新设置编译环境，即需要在编译选项的链接的选项里加上-lwsck32，这样增添的特殊的头文件如：#include<winsock2.h>，#pragma comment(lib, "ws2_32.lib")才能通过编译。

(2) 局域网连接时，一开始服务端和客户端不能实现数据的通信。

答：其实造成不能通信的原因有很多。



代码问题。为验证代码无问题，我们可以采用同一台主机既是客户端又是服务端的方法，将IP改成本机地址。如果服务端能收到数据，证明代码无误。

如果问题还未解决，那么就是IP地址与端口的问题。仔细研究代码，发现服务端和客户端的IP和端口号没有匹配，之前是用手机热点进行的实验，此时的端口地址我们无法确定，因此实验不成功。因此后来采取服务端主机用网线连网，没有经过路由器，客户端使用校园网WIFI，改正之后实验成功，实现了局域网通信，并且统计出来丢包率为0。

(3) 外网访问中大校园网IP受限, 互联网通信难度较大。

答：外网与内网之间的通信，需要经过路由器，一般外网用户访问中大内网，都需要VPN。由于UDP是无连接协议，是无连接协议，客户端的端口是由客户端随机生成的。解决办法是做端口映射。假设我内网的IP为：172.18.154.95，此时和外网的IP：119.23.204.50通信，那么指定固定的IP+UDP端口给内网的客户端，固定端口为：5001，实现自由双向通信。从客户端发起请求，可以根据外网的IP和端口顺利找到服务器，若是服务器给内网的机器发就很难实现，因为我的内网IP和端口在经过我的网关之后，都会发生变化。实际和服务器通信的是网关转换后的地址和端口，内网IP和端口只有网关知道是哪台机器。

(4) 即使搭建好自己的服务器，已知服务器外网IP, 但是也没有实现通信。

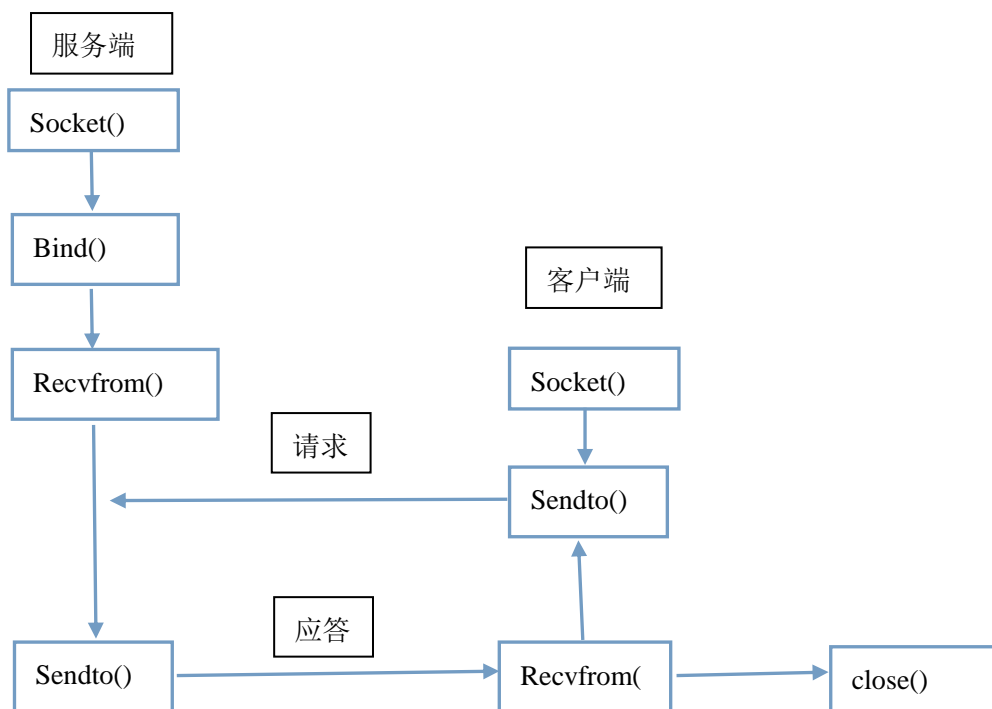
答：服务器要设置好安全组规则，允许UDP协议，开放指定端口，并且注意客户端写的服务端IP地址是其外网地址而不是内网地址。最后实现通信。

(5) 什么时候是用外网地址，什么时候用内网地址？

答：局域网通信时，代码里填写服务端内网地址，最后服务端显示的是客户端的内网地址，互联网通信时要填写服务端外网地址，最后服务端显示的是客户端的外网地址。如果填错显然无法实现通信。

二、给出程序详细的流程图和对程序关键函数的详细说明。

1. 流程图





2. 函数说明

【A. 服务端】

(1). 创建套接字

相关函数: `socket()`

```
SOCKET WINAPI socket(  
    _In_ int af, /*指明使用的协议族*/  
    _In_ int type, /*指明 socket 类型*/  
    _In_ int protocol /*指明要使用的协议*/  
);
```

(2). 将本地IP地址和端口号绑定到套接字

相关函数: `bind()`

```
int bind(  
    _In_ SOCKET s, /*由 socket () 返回的套接口文件描述符*/  
    _In_ const struct sockaddr *name, /*包含地址、端口等信息的数据结构*/  
    _In_ int namelen /*sockaddr 的长度*/  
);
```

(3). 接收数据(请求)

相关函数: `recvfrom()`

```
int recvfrom(  
    _In_ SOCKET s, /*接收端*/  
    _Out_ char *buf, /*将要读取的数据的缓冲区*/  
    _In_ int len, /*缓冲区的最大长度*/  
    _In_ int flags, /*一般设为 0*/  
    _Out_ struct sockaddr *from, /*获取到发送端的地址*/  
    _Inout_opt_ int *fromlen /*sockaddr 的长度*/  
);
```

If no error occurs, **recvfrom** returns the **number of bytes received**. If the connection has been gracefully closed, the return value is **zero**. Otherwise, a value of **SOCKET_ERROR** is returned.

(4). 发送数据(应答)

相关函数 `sendto()`

```
int sendto(  
    _In_ SOCKET s, /*发送端*/  
    _In_ const char *buf, /*要发送的数据的缓冲区*/  
    _In_ int len, /*缓冲区的最大长度*/  
    _In_ int flags, /*一般设为 0*/  
    _In_ const struct sockaddr *to, /*需要指明目的地址*/  
    _In_ int tolen /*sockaddr 的长度*/  
);
```

If no error occurs, **sendto** returns the **total number of bytes sent**, which can be less than the number indicated by *len*. Otherwise, a value of **SOCKET_ERROR** is returned



【B. 客户端】（函数说明同服务端）

1. 创建套接字
2. 发送数据(请求)
3. 接收数据(应答)

三、使用Socket API开发通信程序中的客户端程序和服务器程序时，各需要哪些不同的函数？

在无连接的Client/Server结构中：

客户端：创建socket套接字:socket(); 向服务器发送消息:sendto(); 接收服务端应答消息:recvfrom(); 关闭socket套接字:closesocket()

服务端：创建socket套接字:socket(); 将套接字绑定到指定的IP地址和端口上:bind(); 接收消息:recvfrom(); 向客户端发发送应答:sendto(); 关闭socket套接字:closesocket()

四、解释connect(),bind()等函数中struct sockaddr * addr参数各个部分的含义，并用具体的数据举例说明。

```
struct sockaddr_in {  
    short        sin_family;   //指代协议族，在 socket 编程中只能是 AF_INET  
    unsigned short sin_port;   //使用的端口，2 字节  
    struct in_addr sin_addr;   //IP 地址，4 字节  
    char         sinzero[8];  //为了让 sockaddr 与 sockaddr_in 两个数据结构保持大小相同而保留的空  
                                字节  
};  
  
struct sockaddr {  
    u_short      sa_family;   //地址家族，一般都是“AF_XXX”的形式。  
                                //通常大多用的是都是 AF_INET,代表 TCP/IP 协议族。  
    char         sa_data[14]; // 14 字节协议地址  
};
```

两种数据类型的联系与区别：

1. 这两个结构体一样大，都是16个字节；
2. sockaddr用其余14个字节来表示sa_data，而sockaddr_in把14个字节拆分成sin_port，sin_addr和sin_zero；
3. 使用上有区别，程序员不应操作sockaddr，sockaddr是给操作系统用的；
4. 一般的用法为程序员把类型、ip地址、端口填充sockaddr_in结构体，然后强制转换成sockaddr，作为参数传递给系统调用函数。

具体数据举例说明

```
sockaddr_in example_addr = {  
    AF_INET,           // TCP/IP 协议族  
    htons(6000),        //使用的端口为 6000  
}  
example_addr.sin_addr.S_un.S_addr = htonl("127.0.0.1"); // IP 地址为"127.0.0.1"
```

五、说明面向连接的客户端和面向非连接的客户端在建立Socket时有什么区别？：

都是调用下面的函数建立socket：



```
SOCKET socket(int domain, int type, int protocol);
```

面向连接的客户端: `type = SOCK_STREAM`;

面向非连接的客户端: `type = SOCK_DGRAM`

六、说明面向连接的客户端和面向非连接的客户端在收发数据时有什么区别。面向非连接的客户端又是如何判断数据发送结束的?

面向连接: 客户端和服务端建立连接之后, 通过使用`read()`和`write()`来接收和发送数据;

非面向连接: `sendto()`函数, 在参数中指明目的IP地址和端口号, 向指定方发送数据包。

`recvfrom()`函数从socket收到一个数据包时, 将发送此数据包的进程的网络地址以及数据包本身都保存下来。通过这个保存下来的地址确定发送这个数据包的来源地址;

非连接的客户端可以用`sendto()`函数的返回值来判断数据是否发送结束。若返回实际发送数据的字节大小, 则发送成功; 若返回`SOCKET_ERROR`, 则是发送失败。

七、比较面向连接的通信和无连接通信, 他们各有什么优点和缺点? 适合在哪种场合使用?

面向连接:

优点: 可靠, 无差错, 不丢失, 不重复, 按序到达

缺点: 传输效率低

例子: 电子邮件

无连接:

优点: 不用建立连接, 效率高

缺点: 不可靠, 会出现丢包、重复、乱序

例子: 即时通讯

八、实验过程中Socket是工作在阻塞方式还是非阻塞方式? 通过网络检索阐述这两种操作方式的不同。

实验中是阻塞方式。

```
// 3. 等待客户数据
while(1) {
    // 处理请求
    ..
    // 应答
    ..
}
}
```

阻塞方式是指, 若上一步动作没有完成, 则下一步动作无法进行, 直到上一步动作完成后才可以进行下一步的动作。Windows套接字在阻塞和非阻塞两种模式下执行I/O操作。在阻塞模式下, 在I/O操作完成前, 执行的操作函数一直等候而不会立即返回, 该函数所在的线程会阻塞在这里。相反, 在非阻塞模式下, 套接字函数会立即返回, 而不管I/O 是否完成, 该函数所在的线程会继续运行。

引起UDP丢包的可能原因是什么?

在本次实验中, 我们用IPERF软件对UDP进行检测, 发现在大量传输数据时, 互联网通信会有一定的丢包率。我们小组经过反复的慎重讨论, 究其原因可能有以下几个方面:



计算机网络实验报告

1. 数据传输频率过快，导致丢包。我们小组在互联网通信实验的第一次实验控制的自变量为100个数据，且手动输入，数据传输频率极低，故丢包率为零。后来当我们进行大量高频的数据传输时，丢包的情况也就发生了。当我们了解了UDP协议的原理时才明白，UDP不会像TCP协议那样，将已有的数据全部发送后再返回原来的调用函数，而是直接不造成线程阻塞，一直传输下去，这就导致这期间传送的某些报文可能丢失。这种不要求分组顺序到达的极致，导致了它作为一个简单不可靠信息传送服务可能带来的丢包。
2. 没有限制报文长度。UDP协议有自己的报文最大负载和报文最小长度。我们在使用IPERF传输数据包的时候，是没有限制其报文长度的。一旦报文长度超过了UDP自己的最大负载或者小于最小长度，就极有可能造成数据的丢失。因此，没有对报文长度进行一定限制，也是造成数据丢失的一大因素。经过查阅相关资料我们发现，当发送的数据包过大时，我们可以采用把其切割成小包的方式进行传输。
3. 接收端的相应处理过程耗费时间，二次调用造成丢包。UDP没有线程阻塞，不断地接收数据的过程中，在二次调用的间隔之间会造成一定的丢包可能性。因为我们没有设置缓冲区，所以数据不断大量传输，必然导致丢包。
4. 发送端的相应处理也需要耗费时间，缓冲区小。这种情况下，我们高速大量地发送数据，发送速度过快，而缓冲区的容量又太小，无法缓存大量的数据，可能导致接收方无法接收到全部的数据，造成丢包。

本次实验完成后，请根据组员在实验中的贡献，请实事求是，自评在实验中应得的分数。（按百分制）

学号	学生	自评分
16339021	回煜森	100
16339049	辛依繁	100
16343065	桑娜	100