计算机网络实验报告

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。

专业	计算机科学与技术	班 级	超级计算方向	组长	林天皓
学号	18324034				
学生					

编程 实验

【实验内容】

（1）完成实验教程实例 3-2 的实验（考虑局域网、互联网两种实验环境），回答实验提出的问题及实验思考。（P103）。

（2）注意实验时简述设计思路。

（3）引起 UDP 丢包的可能原因是什么？

【实验需求】

根据实验教程 3-2 ，本次实验需要完成一个 UDP 统计丢包程序。在发送 UDP 数据包的时候做一个循环，连续发送 100 个数据包；在接收端统计丢失的数据包。

【实验过程与结果】

UDP 全称 User Datagram Protocol，UDP 协议是一种传输层协议，UDP 使用具有最少协议机制的简单无连接通信模型。 UDP 提供用于数据完整性的校验和，以及用于在数据报的源和目标处寻址不同功能的端口号。它没有握手对话，因此使用户程序暴露于底层网络的任何不可靠性；不能保证到达，顺序或重复发送。 UDP 适用于不需要或在应用程序中执行错误检查和纠正的目的。 UDP 避免了协议栈中此类处理的开销。对时间敏感的应用程序通常使用 UDP。

UDP 数据包在 IPV4 中的格式定义如下，具有源地址，目的地址，校验和等条目。

IPv4 pseudo header format

Offsets	Octet	0							1							2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source IPv4 Address																															
4	32	Destination IPv4 Address																															
8	64	Zeroes								Protocol								UDP Length														
12	96	Source Port																								Destination Port							
16	128	Length																								Checksum							
20	160+	Data																															

图 1-UDP 报文格式

计算机网络实验报告

本次实验中在 windows 平台下进行，使用 Winsock 2.2 库进行网络编程。首先使用 C++语言分别实现客户端和服务端发送和接收 UDP 数据包的程序。

一、客户端程序

客户端编写的程序流程图如下：

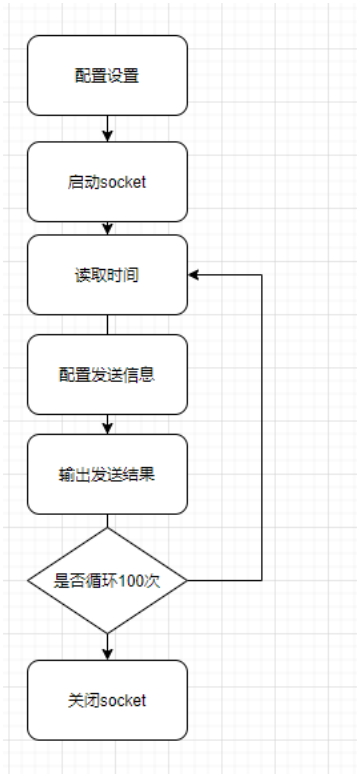

图 2-客户端程序流程图

1.配置设置：在客户端程序中，首先需要指定 socket 的基本配置项目，例如 ip 目的地的地址，ip 目的地的端口，使用的网络通信协议。客户端程序中相关代码如下：

```
struct sockaddr_in socksend;
SOCKET sock;
WSADATA wsadata;
char host []= "172.18.198.217"; //目的地址
u_short serverport = 30011; //目的地址端口
WSAStartup(MAKEWORD(2, 2), &wsadata); //设置 winsock 版本 2.2
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //
memset(&socksend, 0, sizeof(socksend));
socksend.sin_family = AF_INET;
socksend.sin_port = htons(serverport);
socksend.sin_addr.s_addr = inet_addr(host);
```

通过 WSAStartup 指定我们使用的 winsock 协议的版本为 2.2，通过 AF_INET 指定使用的 ip 地址协议为 IPV4。SOCK_DGRAM, IPPROTO_UDP 指定使用的协议为 UDP 协议。

2.读取时间：为了测试使用 UDP 协议的通信延时，我们在客户端发送信息的同时，先读取目前获取到的时间，加入发送的数据内部，以便服务端可以得知发送的时间获取通信时间差。

获取到毫秒时间的函数如下：

```
long long GetCurrentTimeMsec()
{
#ifdef _WIN32
        struct timeval tv;
        time_t clock;
        struct tm tm;
        SYSTEMTIME wtm;
        GetLocalTime(&wtm);
        tm.tm_year = wtm.wYear - 1900;
        tm.tm_mon = wtm.wMonth - 1;
        tm.tm_mday = wtm.wDay;
        tm.tm_hour = wtm.wHour;
        tm.tm_min = wtm.wMinute;
        tm.tm_sec = wtm.wSecond;
        tm.tm_isdst = -1;
        clock = mktime(&tm);
        tv.tv_sec = clock;
        tv.tv_usec = wtm.wMilliseconds * 1000;
        return (tv.tv_sec * 1000 + tv.tv_usec / 1000);
#else
        struct timeval tv;
        gettimeofday(&tv,NULL);
        return ((unsigned long long)tv.tv_sec * 1000 + (unsigned long long)tv.tv_usec / 10
00);
#endif
    }
```

在这段代码中通过 GetLocalTimeu，可以获得毫秒级的当前时间。

3.为了控制发送的速度，这里使用一个延时函数控制。

```
void Delay(int time)
{
    clock_t now = clock();
    while(clock()-now<time);
```

```
}
```

4.发送数据：同时一个循环发送 100 次数据，每次发送的数据中有发送的 id 和发送的时间。

通过 sendto 函数执行对数据的发送。

```
    for(int i=0;i<100;i++){
        long long send_time = GetCurrentTimeMsec();
        sprintf(send_message,"UDP test id:%2d ,sendtime=%lld",i,send_time);
        int sendlen = sendto(sock, send_message, BUFLEN, 0, (SOCKADDR*)& socksend, sizeof(
socksend));
        if (sendlen == SOCKET_ERROR)
        {
            printf("[error] 第%3d 次发送失败",i);
        }
        printf("[info] 第%d 次发送成功 发送内容: %s\n",i,send_message);
        Delay(100);
    }
```

4.结束后关闭 socket，通过调用 closesocket 和 WSACleanup 关闭。

```
    closesocket(sock);
    WSACleanup();
```

客户端的完整代码如下：

```
//#define _CRT_SECURE_NO_WARNINGS
//#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <cstdlib>
#include <cstdio>
#include <cstring>
#include <string>
#ifdef _WIN32
#include <winsock2.h>
#include <windows.h>
    #include <ctime>
#else
    #include <sys/time.h>
#endif
long long GetCurrentTimeMsec()
{
#ifdef _WIN32
        struct timeval tv;
        time_t clock;
        struct tm tm;
        SYSTEMTIME wtm;
```

```
        GetLocalTime(&wtm);
        tm.tm_year = wtm.wYear - 1900;
        tm.tm_mon = wtm.wMonth - 1;
        tm.tm_mday = wtm.wDay;
        tm.tm_hour = wtm.wHour;
        tm.tm_min = wtm.wMinute;
        tm.tm_sec = wtm.wSecond;
        tm.tm_isdst = -1;
        clock = mktime(&tm);
        tv.tv_sec = clock;
        tv.tv_usec = wtm.wMilliseconds * 1000;
        return (tv.tv_sec * 1000 + tv.tv_usec / 1000);
#else
        struct timeval tv;
        gettimeofday(&tv,NULL);
        return ((unsigned long long)tv.tv_sec * 1000 + (unsigned long long)tv.tv_usec / 10
00);
#endif
    }

void Delay(int time)
{
    clock_t now = clock();
    while(clock()-now<time);
}
#define BUFLEN 1000                 // 缓冲区大小
#pragma comment(lib, "ws2_32.lib") // 加载 winsock 2.2 Llibrary

int connectAndSendto();
int main()
{
    printf("[info]\n 开始 udp 丢包率测试\n");
    printf("=======================================\n");

    connectAndSendto();

    printf("运行结束，任意键退出。\n");
    getchar();
    return 0;
}

int connectAndSendto()
{
    struct sockaddr_in socksend;
```

```
    SOCKET sock;
    WSADATA wsadata;
    WSAStartup(MAKEWORD(2, 2), &wsadata); //设置 winsock 版本 2.2
    memset(&socksend, 0, sizeof(socksend));
    char host []= "127.0.0.1"; //目的地址
    u_short serverport = 30011; //目的地址端口
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //
    socksend.sin_family = AF_INET;
    socksend.sin_port = htons(serverport);
    socksend.sin_addr.s_addr = inet_addr(host);
    char send_message[BUFLEN + 1];
    for(int i=0;i<100;i++){
        long long send_time = GetCurrentTimeMsec();
        sprintf(send_message,"UDP test id:%d,sendtime=%lld",i,send_time);
        int sendlen = sendto(sock, send_message, BUFLEN, 0, (SOCKADDR*)& socksend, sizeof(
socksend));
        if (sendlen == SOCKET_ERROR)
        {
            printf("[error] 第%3d 次发送失败",i);
        }
        printf("[info] 第%d 次发送成功 发送内容: %s\n",i,send_message);
        Delay(100);
    }
    closesocket(sock);
    WSACleanup();
    return 1;
}
```

二、服务端程序

服务端编写的程序流程图如下：

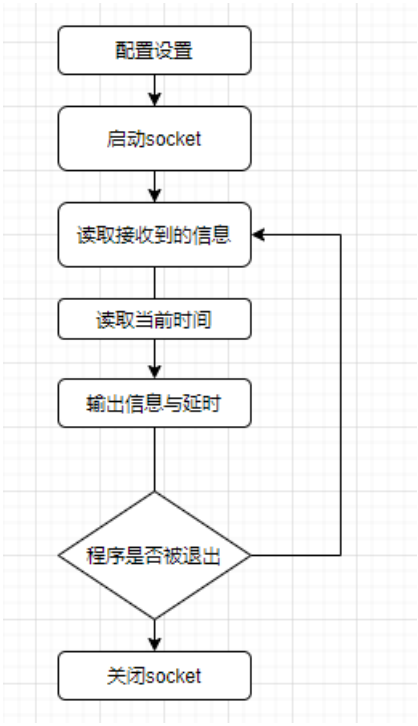

图 3-服务端程序流程图

由于服务端的代码框架与客户端结构基本相同，这里只列出不同的接收的循环结构中的程序。

1.接收端：通过调用 recvfrom 阻塞的接收来自客户端的信息，通过使用 regex 正则表达式查找信息内部的发送时间，计算延时并输出。其中正则表达式为：

```
regex send_time_expr(".*sendtime=(.*?)");
```

该表达式通过匹配并提取接收到的信息中的 sendtime=后的字符串，然后通过 stringstream 的输入与输出完成从字符串到 long long 类型的转换。

循环部分代码如下：

```
while (1)
{
    int recvlen = recvfrom(sock, buf, BUFLEN, 0, (SOCKADDR*)& from, &fromsize);
    long long recv_time = GetCurrentTimeMsec();
    time_t now = time(NULL);
    char* timestr = ctime(&now);
    buf[recvlen] = '\0';
    string recv_msg(buf);
    std::smatch match_msg;
    long long send_time;
    regex_match(recv_msg,match_msg,send_time_expr);
```

```
        if(match_msg.size()>1){
            std::stringstream strstream;
            strstream << match_msg[2].str();
            strstream >> send_time;
            printf("收到第%d 条消息：%s\n", success_count++ ,recv_msg.c_str());
            printf("该条消息的发送-接收时间差为: %lld ms\n",recv_time-send_time);
            if(match_msg[1].str()=="99")
                break;
        }
    }
```

2.关闭 socket 连接并输出丢包率测试

```
    closesocket(sock);
    WSACleanup();
    printf("接收到的数据包数量：%d\n 丢包率：%.3f%%\n",success_count,1.0f*(100-
success_count));
    printf("服务端运行结束\n");
```

服务端的完整代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <winsock2.h>
#include <windows.h>
#include <string.h>
#include <time.h>
#include <conio.h>
#include <sstream>
#include <regex>
using namespace std;
#pragma comment(lib, "ws2_32.lib")
#define BUFLEN 1000
long long GetCurrentTimeMsec()
{
#ifdef _WIN32
        struct timeval tv;
        time_t clock;
        struct tm tm;
        SYSTEMTIME wtm;
        GetLocalTime(&wtm);
        tm.tm_year = wtm.wYear - 1900;
        tm.tm_mon = wtm.wMonth - 1;
        tm.tm_mday = wtm.wDay;
        tm.tm_hour = wtm.wHour;
        tm.tm_min = wtm.wMinute;
```

```
        tm.tm_sec = wtm.wSecond;
        tm.tm_isdst = -1;
        clock = mktime(&tm);
        tv.tv_sec = clock;
        tv.tv_usec = wtm.wMilliseconds * 1000;
        return (tv.tv_sec * 1000 + tv.tv_usec / 1000);
#else
        struct timeval tv;
        gettimeofday(&tv,NULL);
        return ((unsigned long long)tv.tv_sec * 1000 + (unsigned long long)tv.tv_usec / 10
00);
#endif
    }

void Delay(int time)
{
    clock_t now = clock();
    while(clock()-now<time);
}

void makeNewMsg(char* msg, char* timestr, struct sockaddr_in from);
int main()
{
    u_short port = 30011;
    struct sockaddr_in sock_in;
    struct sockaddr_in from;
    int fromsize = sizeof(from);
    SOCKET sock;
    WSADATA wsadata;
    WSAStartup(MAKEWORD(2, 2), &wsadata);
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    memset(&sock_in, 0, sizeof(sock_in));
    sock_in.sin_port = htons(port);
    sock_in.sin_family = AF_INET; //指定 ipv4
    sock_in.sin_addr.s_addr = INADDR_ANY;
    bind(sock, (struct sockaddr*) & sock_in, sizeof(sock_in));

    time_t now = time(NULL);
    printf("[info] UDP 测试服务端启动 \n 服务器启动时间: %s", ctime(&now));

    char buf[BUFLEN + 1];
    int success_count=0;
    regex send_time_expr(".*id:(.*?),.*sendtime=(.*?)");
    while (1)
```

```
    {
        int recvlen = recvfrom(sock, buf, BUFLEN, 0, (SOCKADDR*)& from, &fromsize);
        long long recv_time = GetCurrentTimeMsec();
        time_t now = time(NULL);
        char* timestr = ctime(&now);
        buf[recvlen] = '\0';
        string recv_msg(buf);
        std::smatch match_msg;
        long long send_time;
        regex_match(recv_msg,match_msg,send_time_expr);
        if(match_msg.size()>1){
            std::stringstream strstream;
            strstream << match_msg[2].str();
            strstream >> send_time;
            printf("收到第%d 条消息：%s\n", success_count++ ,recv_msg.c_str());
            printf("该条消息的发送-接收时间差为：%lld ms\n",recv_time-send_time);
            if(match_msg[1].str()=="99")
                break;
        }
    }
    closesocket(sock);
    WSACleanup();
    printf("接收到的数据包数量：%d\n 丢包率：%.3f%%\n",success_count,1.0f*(100-
success_count));
    printf("服务端运行结束\n");
    getchar();
    return 0;
}
```

【实验结果】

首先使用本机网络进行测试。我们模拟两种情况，一是服务端先启动，客户端再启动，应该能接收到所有的数据包。二是在客户端发送了一些数据包之后启动服务端，这时服务端会不能接收到一些数据包。

测试一：本机网络环境，先启动服务端，再启动客户端

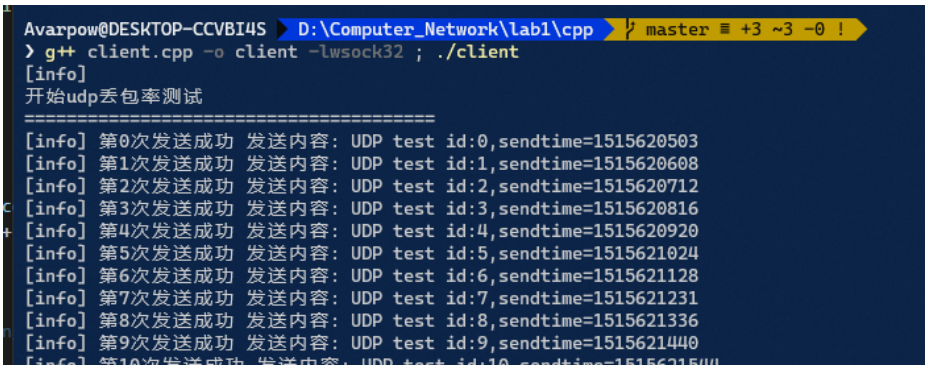

图 4-客户端发送中

分析：输出为客户端发送的次数和信息。

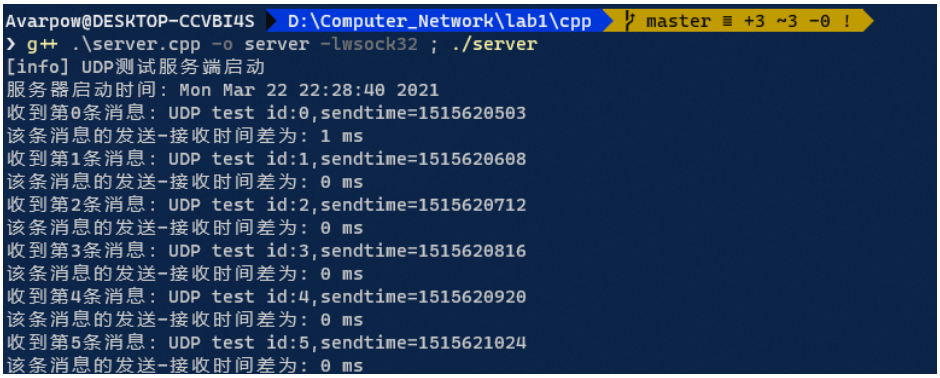

图 5-服务端接收中

分析：输出为服务端接收的发送的次数和信息。

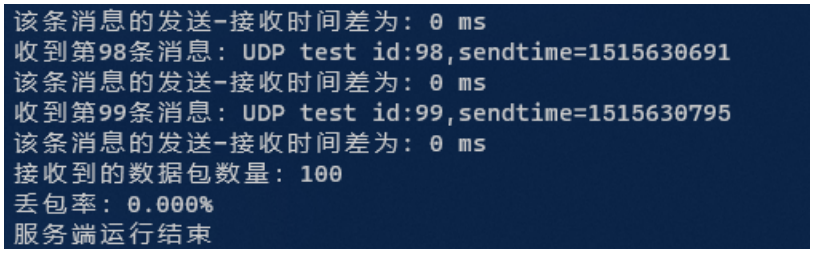

图 6-服务端丢包率 0%

分析：统计丢包率，丢包率为 0%。

测试二：局域网环境测试

客户端 PC：通过校园网 wifi 接入 。服务端 PC：通过校园网有线网接入。

其他内容较为重复，不再赘述此处仅仅展示最后服务端统计信息

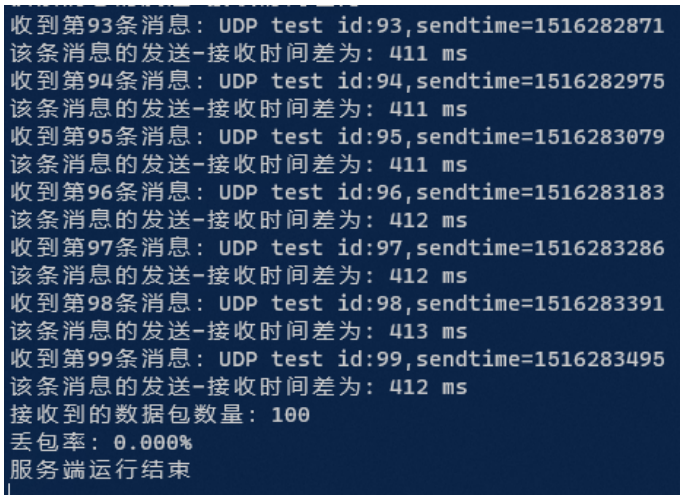

图 7-测试二服务端接收

分析：可见在局域网测试中，通信时间差来到了 400ms，当然这个时间只的不是数据包在链路上的延时，而是包括了发送方封包与客户端解包并提取信息的过程。同时丢包率仍然为 0%。

测试三：互联网环境

服务端 PC：接入校园网内有线网 客户端 PC：广东移动通信，通过校园网 VPN 连接服务端 PC

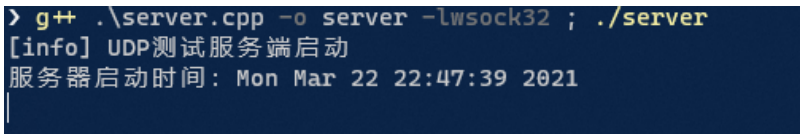

图 8-测试三服务端未能接收到数据

分析：很可惜的是，经过多次尝试，服务端仍然未能接收至少一个到来自客户端的 UDP 数据包。这可能是由于校园 VPN 网对于 UDP 数据包转发的丢弃。

二、使用 wireshark，对通信时的数据包进行跟踪分析

udp

No.	Time	Source	Destination	Protocol	Length	Info
144	11.855555	172.19.26.57	10.8.4.4	DNS	70	Standard query 0xd251 A
145	11.855705	172.19.26.57	10.8.4.4	DNS	70	Standard query 0x3444 AA
146	11.858646	10.8.4.4	172.19.26.57	DNS	86	Standard query response
147	11.858646	10.8.4.4	172.19.26.57	DNS	135	Standard query response
362	27.461643	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
363	27.463219	172.18.198.217	172.19.26.57	ICMP	590	Destination unreachable
364	27.463622	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
365	27.465127	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
366	27.466576	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
367	27.467889	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
368	27.469194	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
369	27.470565	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
370	27.471923	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
371	27.473246	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
372	27.474538	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
373	27.475886	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
374	27.477586	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
375	27.479388	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
376	27.480983	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
377	27.482523	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
378	27.483882	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
379	27.485208	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000
380	27.486534	172.19.26.57	172.18.198.217	UDP	1042	65425 → 30011 Len=1000

图 9-客户端 wireshark 查看数据包

395	27.508072	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
396	27.509526	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
397	27.511122	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
398	27.512894	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
399	27.514364	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
400	27.515732	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001
401	27.517123	172.19.26.57	172.18.198.217	UDP	1042 65425 → 3001

```
Checksum: 0x3d3a [unverified]
[Checksum Status: Unverified]
[Stream index: 2]
[Timestamps]
  [Time since first frame: 0.046429000 seconds]
  [Time since previous frame: 0.001433000 seconds]
UDP payload (1000 bytes)
Data (1000 bytes)
Data: 554450207465737420696 43a33322c73656e6474696d653d313531373738313030360000…
[Length: 1000]
```

```
0000  08 68 8d a5 1e 01 94 e7  0b 0c 69 cb 08 00 45 00   ·h······ ··i···E·
0010  04 04 48 71 00 00 80 11  00 00 ac 13 1a 39 ac 12   ··Hq···· ·····9··
0020  c6 d9 ff 91 75 3b 03 f0  3d 3a 55 44 50 20 74 65   ····u;·· =:UDP te
0030  73 74 20 69 64 3a 33 32  2c 73 65 6e 64 74 69 6d   st id:32 ,sendtim
0040  65 3d 31 35 31 37 37 38  31 30 30 36 00 00 00 00   e=151778 1006····
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ···· ···· ····
```

图 10-验证客户端 wireshark 数据包信息

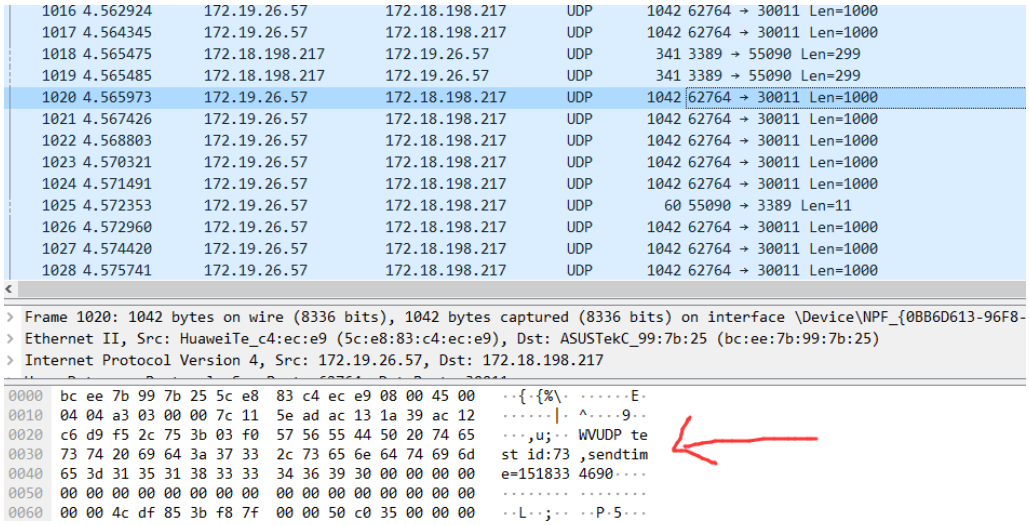

图 11-服务端 wireshark 查看 UDP 数据包信息

分析：通过 wireshark 软件我们可以分别在客户端和服务端查看到发送和接收到的数据包，通过查看内部发送的数据，我们可以看到正是我们通过用户程序所发送的信息。

综上，在本次实验中使用 socket api 完成了 UDP 丢包率测试软件的实现。

【实验思考】

问题解答

（1）说明在实验过程中遇到的问题和解决方法。

答:在本次实验中，需要使用 windows socket api 进行编程，在使用库函数时需要了解新定义的参数，调用方法，结构体的含义等，通过对资料的查阅，逐渐掌握了了对 windows socket api 的使用方法。同时，在进行编译时，还需要添加-lwsock32 编译参数才能编译通过。

（2）给出程序详细的流程图和对程序关键函数的详细说明。

上文中已经给出程序流程图和关键函数说明，此处不再赘述。

（3）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？

首先通过 WSAStartup(MAKEWORD(2, 2), &wsadata); 指定 windows sockets api 的版本使用，再使用 sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);指定使用的网络 ip 地址为 IPV4，协议为 UDP。

对于服务端的程序，我们使用 bind(sock, (struct sockaddr*) & sock_in, sizeof(sock_in));通过 bind 函数将构造的 sock 结构绑定到本机端口上。然后使用 recvfrom(sock, buf, BUFLEN, 0, (SOCKADDR*)& from, &fromsize);从客户端接收网络信息。

对于客户端程序，需要使用 sendto(sock, send_message, BUFLEN, 0, (SOCKADDR*)& socksend, sizeof(socksend));完成对网络信息的发送。

（4）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。

Sockaddr 中，包括 sin_port 表示端口，sin_addr 表示网络的 ip 地址，sa_family 表示地址协议家族，可选 IPV4 或者 IPV6。

```
char host []= "127.0.0.1"; //目的地址
u_short serverport = 30011; //目的地址端口
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP); //
socksend.sin_family = AF_INET;
socksend.sin_port = htons(serverport);
socksend.sin_addr.s_addr = inet_addr(host);
```

在以上的代码中设置了使用 IPV4，地址为 127.0.0.1，端口为 30011.

（5）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。

对于面向连接的客户端，首先在定义 socket 时需要使用参数 SOCK_STREAM。 面向非连接的 socket 指定的参数为 SOCK_DGRAM。

（6）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？

对于面向连接的客户端，需要先使用 connect 函数连接到服务器，才能执行 send 进行信息的发送。

对于面向非连接的客户端，不需要需要先使用 connect 函数连接到服务器，可以直接执行 send 进行信息的发送。

（7）比较面向连接的通信和无连接通信，它们各有什么优点和缺点？适合在哪种场合下使用？

无连接通信没有握手对话，因此使用户程序暴露于底层网络的任何不可靠性；不能保证到达， 顺序或重复发送。无连接通信适用于不需要或在应用程序中执行错误检查和纠正的目的。而有连接通信保证实时通信，可以接收到对方的回复，保证数据的到达。需要连接可靠的服务使用面向连接的通信。对时间敏感的应用程序通常使用无连接通信。

（8）实验过程中使用 Socket 时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。

使用的是阻塞方式。调用结果返回之前，当前线程被挂起。函数只有得到结果才返回。非阻塞 方式即使不能立刻得到结果之前，该函数不会阻挡当前线程，会立刻返回。

【实验总结】

在本次实验中，通过学习计算机网络中 UDP 网络相关原理，了解了如何使用 socket 库中的配置与绑定端口，实现一个多机之间通过 UDP 通信的程序，并且可以统计丢包率。在此基础上了解了 windows 和 linux 下如何获取毫秒级的时间戳，设定延时函数，并查看了使用 socket 传输过程中造成的延迟，经过对比发现越高的发送速度会造成越大的延迟。

除了使用基本的 C++语言调用 winsock 库函数进行编程之外，还尝试使用了 go 语言进行编程。Go 语言原生支持协程等特性，能有效快速的编写高性能的网络应用程序。在 go 语言中实现的服务端代码如下：

```
package main

import (
    "fmt"
    "net"
    "regexp"
```

```
    "time"
    "strconv"
)

// UDP Server 端
func main() {
    reg1 := regexp.MustCompile(`.*id:(.*?),.*sendtime=(.*?)ms`)
    if reg1 == nil {
        fmt.Println("regexp err")
        return
    }
    listen, err := net.ListenUDP("udp", &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30011,
    })
    if err != nil {
        fmt.Println("Listen failed, err: ", err)
        return
    }
    cont:=0
    defer listen.Close()
    fmt.Println("服务器启动", err)
    for {
        var data [1024]byte
        _, _,err := listen.ReadFromUDP(data[:]) // 接收数据
        t := time.Now().UnixNano() / 1e6
        if err != nil {
            fmt.Println("read udp failed, err: ", err)
            continue
        }
        result1 := reg1.FindStringSubmatch(string(data[:]))
        fmt.Printf("收到第%d 条消息：%s\n",cont, string(data[:]))
        cont=cont+1
        sen, _ := strconv.ParseInt(result1[2], 10, 64)
        fmt.Printf("该条消息的发送-接收时间差为:%vms\n", sen-t)
        if result1[1]=="99" {
            break
        }
    }
    fmt.Printf("接收到的数据包数量：%d\n 丢包率：%.3f%%\n",cont,float32(100-cont));
    fmt.Printf("服务端运行结束\n");
}
```

客户端的代码如下：

```go
package main

import (
    "os"
    "fmt"
    "net"
    "time"
)

func main() {
    conn, err := net.Dial("udp", "172.18.198.217:30011")
    defer conn.Close()
    if err != nil {
        os.Exit(1)
    }
    for i:=0; i<100;i++ {
        t := fmt.Sprintf("%v",time.Now().UnixNano() / 1e6);
        s := fmt.Sprintf("UDP test id:%d,sendtime=%vms",i,t)
        fmt.Printf("发送第%v 条消息:",i)
        fmt.Printf(s+"\n")
        conn.Write([]byte(s))
        time.Sleep(time.Duration(1)*time.Millisecond)
    }
    fmt.Println("发送信息完成")

}
```

下面在局域网环境中进行测试。

```
Avarpow@DESKTOP-CCVBI4S  D:\Computer_Network\lab1\go  master ≡
> go run .\server.go
服务器启动 <nil>
收到第0条消息： UDP test id:0,sendtime=1616470224694ms
该条消息的发送-接收时间差为:71ms
收到第1条消息： UDP test id:1,sendtime=1616470224702ms
该条消息的发送-接收时间差为:71ms
收到第2条消息： UDP test id:2,sendtime=1616470224712ms
该条消息的发送-接收时间差为:73ms
收到第3条消息： UDP test id:3,sendtime=1616470224722ms
该条消息的发送-接收时间差为:73ms
收到第4条消息： UDP test id:4,sendtime=1616470224732ms
该条消息的发送-接收时间差为:70ms
收到第5条消息： UDP test id:5,sendtime=1616470224742ms
该条消息的发送-接收时间差为:73ms
收到第6条消息： UDP test id:6,sendtime=1616470224752ms
该条消息的发送-接收时间差为:72ms
```

图 12-服务端接收 UDP 数据包代码

图 13-服务端计算 UDP 丢包率

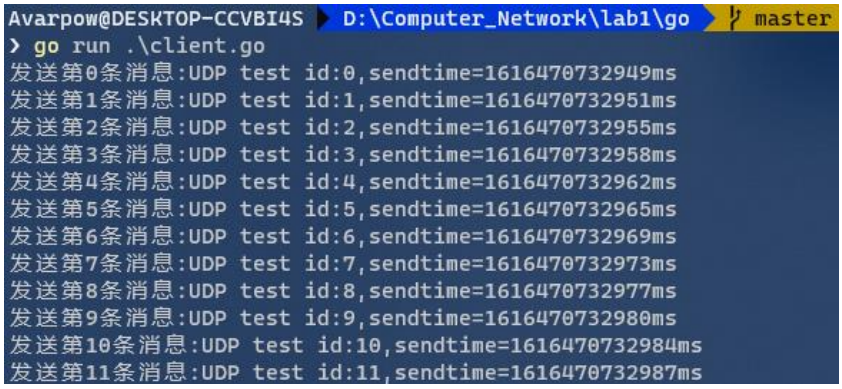

图 14-客户端发送 UDP 数据包状态

通过对比 C++语言的代码，go 语言写的代码更加简洁，能到达同样的效果，同时对于代码安全性的检查也更加完善。