

1. 实验目的

- (一) 熟悉 UDP 数据包格式
- (二) 掌握 UDP 数据包的发送和接收处理过程

2. 实验环境

- (一) 物理机: macOS Mojave
- (二) 虚拟机: Ubuntu 18.04
- (三) IDE: VS Code

3. 实验内容

3.1 实验任务

用 C 语言实现 UDP 数据包的发送和接收处理过程, 在 linux 环境下编译程序, 在物理机上发送 udp 数据包并用 wireshark 观察发包收包情况。

3.2 实验过程

3.2.1 代码实现

➤ udp_out 函数

● 函数功能

处理一个要发送的数据包, 实现 UDP 数据包发送的处理。

● 具体思路

- ① 调用 buf_add_header()函数增加 UDP 头部长度空间, 增加长度为 UDP 报头长度——8 字节。
- ② 填写 UDP 报头字段: 源端口号和目的端口号为输入参数, 总长度为 buf 的长度, 校验和字段先填充为 0。注意这四个字段都需要大小端转换。
- ③ 将 UDP 报头字段复制到 buf 中, 然后调用 udp_checksum()函数来计算校验和, 大小端转换之后再填入头部校验和字段。
- ④ 调用 ip_out()函数将 udp 数据报发往 ip 层, 目的 ip 地址由输入参数给出, 上层协议为 UDP 协议

● 代码

```
/**
 * @brief 处理一个要发送的数据包
 * 你首先需要调用buf_add_header()函数增加UDP头部长度空间
 * 填充UDP首部字段
 * 调用udp_checksum()函数计算UDP校验和
 * 将封装的UDP数据报发送到IP层。
 */
/*
 * @param buf 要处理的包
 * @param src_port 源端口号
 * @param dest_ip 目的ip地址
 * @param dest_port 目的端口号
 */
void udp_out(buf_t *buf, uint16_t src_port, uint8_t *dest_ip, uint16_t dest_port)
{
    buf_add_header(buf, 8);
    udp_hdr_t header;
    header.src_port = swap16(src_port);
    header.dest_port = swap16(dest_port);
    header.total_len = swap16(buf->len);
    header.checksum = swap16(0);
    memcpy(buf->data, &header, sizeof(udp_hdr_t));
    uint16_t checksum = udp_checksum(buf, net_if_ip, dest_ip);
    header.checksum = swap16(checksum);
    memcpy(buf->data, &header, sizeof(udp_hdr_t));
    ip_out(buf, dest_ip, NET_PROTOCOL_UDP);
    return;
}
```

端

➤ **udp_in 函数**

● 函数功能

处理收到的 UDP 数据包

● 具体思路

- ① 检查 UDP 报头长度，若长度小于 8，说明该报文不完整，不处理。若完整，转 2。
- ② 计算校验和：先将 udp 报头的校验和缓存起来，然后将其校验和字段清零。调用 `udp_checksum()` 函数计算校验和，若计算结果与缓存的校验和不相等，说明出错，不处理该数据报。若相等，转 3。
- ③ 遍历 `udp_table`，若某个项有效且其端口号等于该报文中的目的端口号字段，说明存在对应的处理函数，转 4。若不存在，转 5。
- ④ 调用 `buf_remove_header()` 函数去掉 udp 报头，然后调用对应的处理函数。
- ⑤ 调用 `buf_add_header()` 函数增加 ip 头部，然后调用 `icmp_unreachable()` 函数发送一个端口不可达的 ICMP 差错报文。

注意：这里只需要增加 ip 头部缓冲区而不需要填写 ip 头部的原因是在调用 `udp_in()` 函数之前，在 `ip_in()` 函数中先用 `buf_remove_header()` 函数移动了 data 指针去掉了 ip 头部，此时我们只需要将指针移回来就可以了。此外，添加 ip 头部的原因是因为 `icmp_unreachable()` 函数需要 ip 头部。

● 代码

```
void udp_in(buf_t *buf, uint8_t *src_ip)
{
    // 检查UDP报头长度
    if (buf->len < 8)
    {
        // 若长度小于8字节，不处理
        return;
    }

    udp_hdr_t *udp = (udp_hdr_t *)buf->data;
    uint16_t checksum = swap16(udp->checksum); // 缓存UDP首部的checksum
    udp->checksum = swap16(0); // 将UDP首部的checksum字段清零
    if (checksum != udp_checksum(buf, src_ip, net_if_ip))
    {
        // 计算后的校验和与之前缓存的checksum不相等，则不处理该数据报
        return;
    }

    for (int i = 0; i < UDP_MAX_HANDLER; i++)
    {
        if (udp_table[i].valid && udp_table[i].port == swap16(udp->dest_port))
        {
            // 若存在对应的处理函数
            buf_remove_header(buf, 8); // 去掉UDP报头
            udp_table[i].handler(&udp_table[i], src_ip, swap16(udp->src_port), buf);
            return;
        }
    }

    // 若不存在，增加IP头部
    buf_add_header(buf, 20);
    // 调用icmp_unreachable()函数发送一个端口不可达的ICMP差错报文
    icmp_unreachable(buf, src_ip, ICMP_CODE_PORT_UNREACH);
    return;
}
```

➤ **udp_checksum 函数**

● 函数功能

计算 udp 数据报的校验和

● 具体思路

- ① 调用 `buf_add_header()` 函数增加 ip 头部，然后将 IP 头部缓存
- ② 调用 `buf_remove_header()` 函数使数据块 `buf` 减少 8 个字节，因为 UDP 伪头部只需要 12 字节，而之前增加 ip 头部时增加了 20 字节。
- ③ 填写 UDP 伪头部: 目的 ip 地址和源 ip 地址由输入参数给出; placeholder 字段置 0; 协议字段为 UDP 协议，即 17; 总长度字段为 `buf` 的长度减去伪头部的长度 12 字节，注意该字段要大小端转换。
- ④ 调用 `checksum16()` 函数计算校验和，计算范围为 udp 伪头部、udp 头部和 udp 数据部分，所以计算长度是 `buf` 的长度。
- ⑤ 调用 `buf_add_header()` 函数增加 8 字节，将之前暂存的 IP 头部拷贝回来。再调用 `buf_remove_header()` 函数只留下 udp 数据报。

● 代码

```
static uint16_t udp_checksum(buf_t *buf, uint8_t *src_ip, uint8_t *dest_ip)
{
    // ip层调用udp_in之前调用buf_remove_header去掉了ip报头
    // 这里调用buf_add_header将ip报头加回来
    buf_add_header(buf, 20);
    ip_hdr_t ip_header;
    memcpy(&ip_header, buf->data, 20); // 将IP头部暂存
    buf_remove_header(buf, 8); // 因为UDP伪头部只需要12字节
    udp_peso_hdr_t udp_peso_header;
    memcpy(udp_peso_header.src_ip, src_ip, 4);
    memcpy(udp_peso_header.dest_ip, dest_ip, 4);
    udp_peso_header.placeholder = 0;
    udp_peso_header.protocol = NET_PROTOCOL_UDP;
    udp_peso_header.total_len = swap16(buf->len - 12);
    memcpy(buf->data, &udp_peso_header, sizeof(udp_peso_hdr_t));
    uint16_t checksum = checksum16((uint16_t *)buf->data, buf->len);
    // 将暂存的IP头部拷贝回来
    buf_add_header(buf, 8);
    memcpy(buf->data, &ip_header, 20);
    // 去掉UDP伪头部
    buf_remove_header(buf, 20);
    return checksum;
}
```

3.2.2 实验自测环境搭建

➤ 修改 `config.h` 中的物理网卡名称和自定义网卡 ip 地址

通过在虚拟机上的终端输入 `ifconfig` 命令行，可以看到物理网卡名称为 `enp0s5`，虚拟机的 ip 地址为 `10.211.55.8`。前三位为网络号，最后一位为主机号。

```
parallels@parallels-Parallels-Virtual-Platform: ~
File Edit View Search Terminal Help
parallels@parallels-Parallels-Virtual-Platform:~$ ifconfig
enp0s5: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.211.55.8 netmask 255.255.255.0 broadcast 10.211.55.255
    inet6 fdb2:2c26:f4e4:0:3f37:5d07:a82b:d53b prefixlen 64 scopeid 0x0<gl
obal>
    inet6 fe80::a2c9:1480:d983:36d6 prefixlen 64 scopeid 0x20<link>
    inet6 fdb2:2c26:f4e4:0:e0ac:cb93:52a6:a7b5 prefixlen 64 scopeid 0x0<gl
obal>
    ether 00:1c:42:f0:6b:61 txqueuelen 1000 (Ethernet)
    RX packets 617635 bytes 615318524 (615.3 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 263723 bytes 227716562 (227.7 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 271242 bytes 298130741 (298.1 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 271242 bytes 298130741 (298.1 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

parallels@parallels-Parallels-Virtual-Platform:~$
```

config.h 中的网卡 ip 地址应该和虚拟机的 ip 地址的网络号相同，但主机号不同。我采用 10.211.55.88。

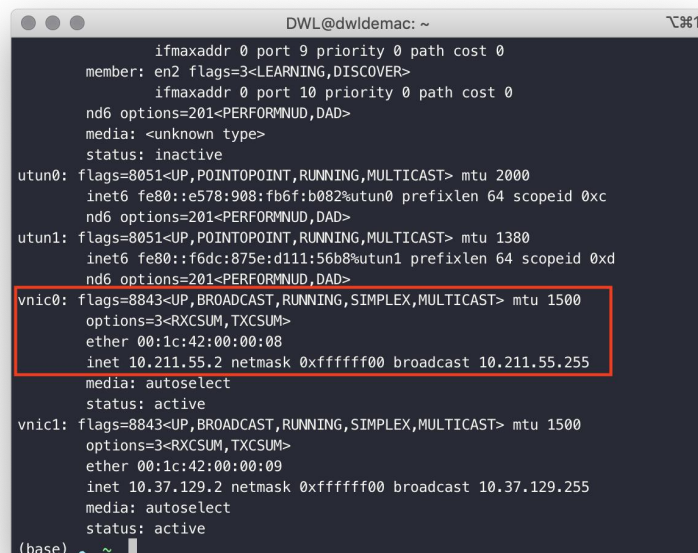
```
#define DRIVER_IF_NAME "enp0s5" //使用的物理网卡名称
#define DRIVER_IF_IP \
{ \
    10, 211, 55, 88 \
} //自定义网卡ip地址
// 前三位为网卡号
// 最后一位为主机号
// 虚拟机IP为10.211.55.8
// 实验中改为10.211.55.88
// 192.168.163.103
```

➤ 编译并运行程序

用 VS Code 中的 cmake 工具对整个工程进行编译，然后右键点击 main，则开始运行。

➤ 打开 wireshark

在物理机的终端中输入 ifconfig -a 命令行语句，可以看到物理机通过 vnic0 和虚拟机连接，ip 为 10.211.55.2，他们在同一个子网下。所以我们打开 wireshark 选择网卡 vnic0 查看抓包结果。



```
DWL@dwidemac: ~
ifmaxaddr 0 port 9 priority 0 path cost 0
member: en2 flags=3<LEARNING,DISCOVER>
ifmaxaddr 0 port 10 priority 0 path cost 0
nd6 options=201<PERFORMNUD,DAD>
media: <unknown type>
status: inactive
utun0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 2000
inet6 fe80::e578:908:fb6f:b082%utun0 prefixlen 64 scopeid 0xc
nd6 options=201<PERFORMNUD,DAD>
utun1: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 1380
inet6 fe80::f6dc:875e:d111:56b8%utun1 prefixlen 64 scopeid 0xd
nd6 options=201<PERFORMNUD,DAD>
vnic0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
options=3<RXCSUM,TXCSUM>
ether 00:1c:42:00:00:08
inet 10.211.55.2 netmask 0xfffff00 broadcast 10.211.55.255
media: autoselect
status: active
vnic1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
options=3<RXCSUM,TXCSUM>
ether 00:1c:42:00:00:09
inet 10.37.129.2 netmask 0xfffff00 broadcast 10.37.129.255
media: autoselect
status: active
(base) ~
```

然后在搜索中选择 ip.src == 10.211.55.88 || ip.dst == 10.211.55.88 的报文，即虚拟机上运行的 udp 程序收到的或者发出的报文。

➤ UDP 调试工具

由于实验所用的物理机为 macOS，无法使用实验所提供的 TCP&UDP 测试工具，所以改用了 netcat 和 hexdump。

netcat 是一款命令行网络测试工具，我将在本次实验中使用 netcat 来发送 udp 报文。安装好 netcat 之后，我在终端中输入 netcat -u 10.211.55.88 60000 -p 60001，u 表示发送的是 udp 报文，10.211.55.88 为目的 ip 地址，60000 为目的端口号，-p 60001 为源端口号。回车之后，可以输入字符串并回车来表示发送 udp 报文。netcat 不仅可以发送 udp 报文，还可以监听本机的端口来接收报文。

但 netcat 将接收的报文以 ASCII 码显示，我们需要将其转换为 16 进制表示。所以采用了 mac 自带的 hexdump 工具。将之前的命令行语句改为 netcat -u 10.211.55.88 60000 -p 60001 | hexdump -C 就可以将收到的报文以 16 进制展示。

还有一点需要注意的是，**hexdump** 会在字符串之后自动加上回车符，所以当我们输入 **abc** 的时候，会显示长度为 4 的报文数据。

3.2.3 实验结果

用 **hexdump** 连续发送两条报文，程序运行结果如下（注意 **hexdump** 会在字符串之后自动加上回车符，所以当我们输入 **abc** 的时候，会显示长度为 4 的报文数据）：

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/net_lab/test# /home/parallels/Desktop/net_lab/test/main
recv udp packet from 10.211.55.2:60001 len=4
abc

recv udp packet from 10.211.55.2:60001 len=4
abc

```

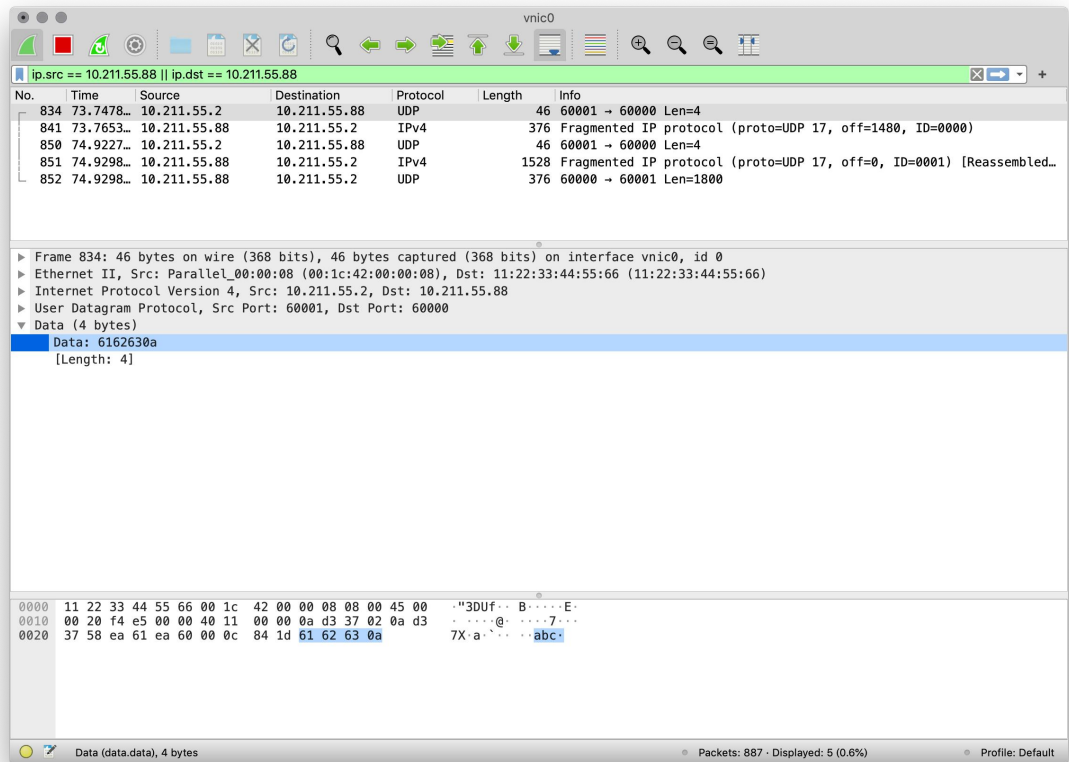
收到的报文如下：

```

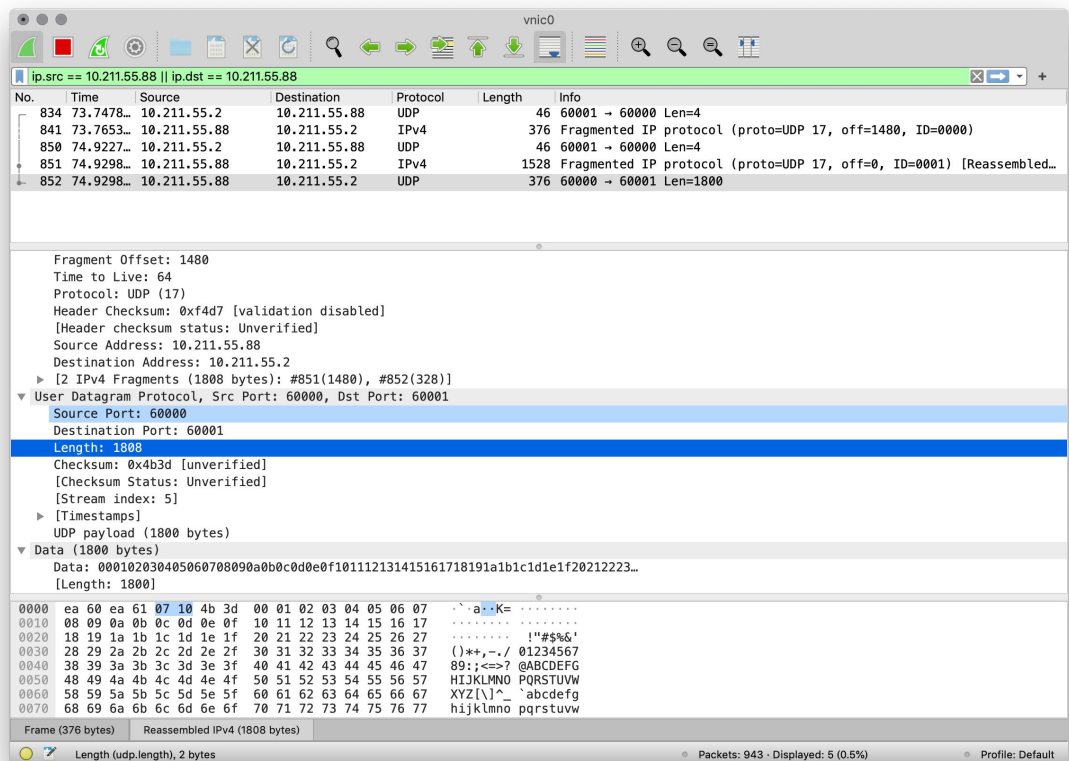
netcat -u 10.211.55.88 60000 -p 60001 | hexdump -C
(base) ~ netcat -u 10.211.55.88 60000 -p 60001 | hexdump -C
abc
abc
00000000  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
00000010  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f |.....|
00000020  20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f | !"#$%&'()*+,-./|
00000030  30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f |0123456789:;<=>?|
00000040  40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f |@ABCDEFGHIJKLMNO|
00000050  50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f |PQRSTUVWXYZ[\]^_|
00000060  60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f |`abcdefghijklmnopqrstuvwxyz{|}~.|
00000070  70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f |pqrstuvwxyz{|}~.|
00000080  80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f |.....|
00000090  90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f |.....|
000000a0  a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af |.....|
000000b0  b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf |.....|
000000c0  c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf |.....|
000000d0  d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df |.....|
000000e0  e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef |.....|
000000f0  f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff |.....|
00000100  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
00000110  10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f |.....|
00000120  20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f | !"#$%&'()*+,-./|
00000130  30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f |0123456789:;<=>?|
00000140  40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f |@ABCDEFGHIJKLMNO|
00000150  50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f |PQRSTUVWXYZ[\]^_|

```

wireshark 抓包结果如下：



根据上图, 可以看到从物理机发到虚拟机的是 **udp** 报文, 报文数据部分长度为 4。



虚拟机第一次收到 **udp** 数据报之后, 也要发送 **udp** 数据报 (长度为 1808 字节),

调用 `send_udp` 函数，然后调用 `udp_out` 函数，然后调用 `ip_out` 函数，然后需要分片发送，调用 `ip_fragment_out` 发送 1480 字节。但是由于 `arp` 表中没有地址，先将其存到 `arp_buf` 里面，然后调用 `arp_req`。但是程序运行为单线程，会继续处理接下来的 `ip` 分片，而不会处理收到的 `arp_reply`。那么剩下的 328 字节又调用 `ip_fragment_out`，再次存到 `arp_buf`。等 `ip_out` 结束之后进行下一次轮询，才处理收到的 `arp_reply`，然后发现 `arp_buf` 是有效的待发送的，就会发出 `buf`，328 字节。这也是为什么物理机要发送两次 `udp` 报文才能收到一次协议栈发来的回复。