ipv4 协议收发实验报告

黄道吉

2020 年 4 月 17 日

IPv4 协议是互联网的核心协议。这次实验要求实现 IPv4 协议中收发数据包的部分。这既包括实现 IPv4 协议向上一层协议提供的接口：接受来自高一层的数据包，按照 IPv4 进行封装，交给下一层；也包括实现本层基本的接受处理功能：接受本层的分组，检查其正确性，正确的转发给上层，否则报错丢弃。

1 背景知识

这次实验的中心是理解 IPv4 协议的报文组成，主要是报头的数据格式。我在实现的过程中主要参考了 RFC791 和 RFC1071，前者定义了 IPv4 报文的结构，后者提供了计算 header checksum 的方法。下文会简要介绍本文用到的部分。

1.1 IPv4 报头格式

下图描述了 IPv4 报头的格式。本次实验关心的是以下部分

- version: 对于 IPv4，version = 4
- IHL: 是指报头长度，以 word(32 bits) 为单位。对于一个有效的报头，IHL 最小为 5，最长的报头只有 15 个 words(RFC791 在 total length 处提到)，本次实验并不考虑 option 部分，所以发送报文时报头长度为 5。
- total length: 整个报文的长度，以 Byte(8 bits) 为单位，包含报头和数据。
- TTL: 报文剩余存货的时间。如果这个域为 0，接受时应当丢弃。
- header checksum: 只对报头部分计算 checksum，计算方法在下面有介绍。
- source/destination address: 发送和接受地址。如果接收地址为本机地址或广播地址，则应接受。

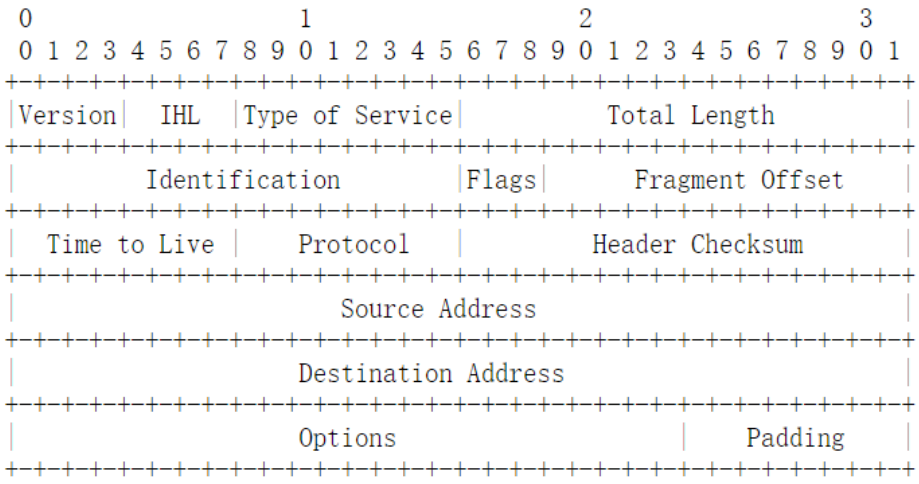

图 1: ipv4 header, fig from RFC791

1.2 checksum 算法

internet checksum 算法在 RFC1071 中有定义和参考的实现。我在实现过程中参考了这一部分的内容。checksum 算法将报头按 16bits 一组分开，分别相加，并将结果的进位部分加到最低位，取反的结果填充到报头中的 checksum 部分。这样填充好的报头再次进行 checksum 操作的时候应当结果为 0。在进行 checksum 时有几个地方是需要注意的

- 初始时 checksum 部分应当为 0
- 计算 checksum 和填充 checksum 时，因为牵扯到两个字节一组，对于组内字节的顺序应使用同一种大小端字节序，采用哪一个字节序对结果没有影响。我的实现使用大端字节序，即报文的第一个 16bits 在计算时看作 $version \times 2^{12} + IHL \times 2^8 + type\ of\ service$，并且 checksum 的高位存储在报头的 10 字节 (从 0 计数)，低位在 11 字节。
- 将进位加到最低位的过程可能需要不止一次，例如 checksum 中间结果为 0x0001FFFF 时，应当进行两次操作。

2 实现细节

本节介绍我的实现的细节和只参考手册和查阅资料没有解决的问题。

2.1 接受报文

检查报文合法性阶段需要分别检验上文提到的各个域，其间的变量最好全部取成无符号整数，以便移位操作不需要考虑符号的问题。在检查目的地址时，如果目的地址是本机地址则必须接受，但广播地址并没有很好的定义，除了 255.255.255.255 以外，回环地址 127.0.0.0/8 和链路本地地址 169.254.0.0/16 是否也需要考虑进去。这可能需要在线上调试的过程中检查。RFC 中定义的 checksum 的过程有考虑字节数为奇或偶的情况，我们的设定中报头都是以 word 为最小长度单位的，不存在奇字节的情况。

2.2 发送报文

发送报文时，因为这次试验不考虑 option 域，报头长度为 5。因为不清楚编译代码的机器是大端还是小端序，因此保险起见按字节填充报头的各个域。

3 代码实现

```
#include <memory.h>

/* API provided by system */

void ip_DiscardPkt(char * pBuffer ,int type);
void ip_SendtoLower(char *pBuffer ,int length);
void ip_SendtoUp(char *pBuffer, int length);
unsigned int getIpv4Address();

int stud_ip_recv(char * pBuffer, unsigned short length){

    // in case version == 1xxx, which would also yield VERSION_ERROR, but just to be concise
    unsigned int version = ((unsigned int)pBuffer[0]) >> 4;
    if(version != 4){
        ip_DiscardPkt(pBuffer, STUD_IP_TEST_VERSION_ERROR);
        return -1;
    }

    unsigned int ip_head_length = ((unsigned int)pBuffer[0]) & 0xF;
    if(ip_head_length < 5 || ip_head_length > 15){
        // according to RFC791, ip_head_length <= 15
        // see RFC791 section 3.1, mentioned in "total length" part
        // ref: https://tools.ietf.org/html/rfc791
        ip_DiscardPkt(pBuffer, STUD_IP_TEST_HEADLEN_ERROR);
        return -1;
    }

    unsigned int time_to_live = ((unsigned int)pBuffer[8]);
    if(time_to_live == 0){
        ip_DiscardPkt(pBuffer, STUD_IP_TEST_TTL_ERROR);
        return -1;
    }

    unsigned int destination_address = ntohl(*(unsigned int* )(pBuffer + 16)); // might be wrong
    if(destination_address != getIpv4Address() || destination_address != (~0)){
        // if it is necessray to consider
        //     - localhost address: 127.0.0.0/8
        //     - link-local address: 169.254.0.0/16
        // then check: (dst_addr >> 24) != 127 and (dst_addr >> 16) != ((169 << 8) + 254)
        ip_DiscardPkt(pBuffer, STUD_IP_TEST_DESTINATION_ERROR);
        return -1;
    }

    unsigned int header_checksum = 0;
    for(int i = 0; i < 4 * ip_head_length; i += 2){
        // 4 * IHL % 2 == 0, no need to consider odd/even issue
        header_checksum += (pBuffer[i] << 8) + pBuffer[i + 1];
    }
    while(header_checksum >> 16){
```

```
        // in case header_checksum == 0x1FFFF
        // add once might not be enough
        // ref: https://tools.ietf.org/html/rfc1071 section 4.1

        // we stick to big-endian byte order, since checksum is byte order independent
        // ref: https://tools.ietf.org/html/rfc1071 section 1.2.(B)
        header_checksum == (header_checksum >> 16) + (header_checksum & 0xFFFF);
    }
    header_checksum = ~header_checksum;

    if(header_checksum != 0){
        ip_DiscardPkt(pBuffer, STUD_IP_TEST_CHECKSUM_ERROR);
        return -1;
    }

    ip_SendtoUp(pBuffer, length);

    return 0;
}

int stud_ip_Upsend(char* pBuffer, unsigned short len, unsigned int srcAddr,
    unsigned int dstAddr ,byte protocol, byte ttl){

    unsigned int version = 4;
    unsigned int ip_head_length = 5; // no option header in this case
    unsigned int total_length = 4 * ip_head_length + len;


    unsigned char* total_buffer = malloc(total_length);
    memset(total_buffer, 0, total_length);

    total_buffer[0] = (version << 4) | ip_head_length;
    // type of service not specified -> set to zero
    total_buffer[2] = total_length >> 8;
    total_buffer[3] = total_length & 0xFF;
    // identification, flags, fragment offset not specified
    total_buffer[8] = time_to_live;
    total_buffer[9] = protocol;

    total_buffer[12] = srcAddr >> 24;
    total_buffer[13] = srcAddr >> 16;
    total_buffer[14] = srcAddr >> 8;
    total_buffer[15] = srcAddr & 0xFF;

    total_buffer[16] = dstAddr >> 24;
    total_buffer[17] = dstAddr >> 16;
    total_buffer[18] = dstAddr >> 8;
    total_buffer[19] = dstAddr & 0xFF;

    unsigned int header_checksum = 0;
    for(int i = 0; i < 20; i += 2){
        header_checksum += (total_buffer[i] << 8) + total_buffer[i + 1];
    }
    while(header_checksum >> 16){
        header_checksum == (header_checksum >> 16) + (header_checksum & 0xFFFF);
    }
    header_checksum = ~header_checksum;
    total_buffer[10] = header_checksum >> 8;
```

```
    total_buffer[11] = header_checksum & 0xFF;

    memcpy(total_buffer + 20, pBuffer, len);

    ip_SendtoLower(total_buffer, total_length);

    return 0;
}
```