计 算 机 网 络
本 科 实 验 报 告

实验名称：TCP/IP 协议栈 ARP 协议实现实验

学 员 姓 名	王李烜	学 号	202202001046
培 养 类 型	无军籍	年 级	2022
专 业	网络工程	所 属 学 院	计算机学院
指 导 教 员	邱振宇	职 称	讲师
实 验 室	305-505	实 验 时 间	2024.12.16

国防科技大学教育训练部制

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

目录

图目录

1 实验概要

1.1 实验内容

本次实验的主要内容 ARP 协议实现。本次实验包含基础任务和拓展任务两部分，具体任务要求如下：

- 基础任务：编写程序，完善 TCP/IP 协议栈的 ARP 协议部分。围绕 ARP 的初始化、无回报 ARP 的生成、ARP 的输入处理，以及 ARP 的超时重新请 求几个部分完成。并且保证完成 ARP 协议的完整实现。本实验中 Completed!
- 拓展任务：拓展任务是可选任务，在基础任务实现的 ARP 协议实现基础上， 可选择性的完成如下任务：
  1. ARP 多个表项的实现；本实验中 Completed!
  2. IP 层的输入输出处理。本实验中 Completed!

1.2 实验要求

本实验的具体过程及对应要求如下：

- 实验开始前准备工作：在实验开始前，学员需要掌握 C 语言 编程基础，理解 TCP/IP 协议栈的工作原理，尤其是 ARP 协议的功能和作用。同时，熟悉 MAC 地址与 IP 地址的转换原理，了解网络设备如何通过 ARP 请求与响应进行地址解析。
- 实验过程中：按照实验要求，完成 ARP 协议的实现。具体步骤包括：具体而言，构造 ARP 请求和响应报文，实现报文格式的编码与解析。发送 ARP 请求，构建并广播 ARP 请求，获取目标设备的 MAC 地址。处理 ARP 响应，在收到响应后，提取并记录目标 IP 与 MAC 地址的映射。管理 ARP 缓存，设计缓存机制，存储 IP-MAC 映射，并实现超时处理机制。
- 实验结束后：总结 ARP 协议的实现过程，详细描述报文格式、缓存管理和通信流程，并根据实验要求撰写实验报告，分析实验结果。

1.3 实验目的

在现代网络环境中， ARP 协议广泛应用于各种网络设备和系统，如计算机、路由器和交换机等。深入理解 ARP 的工作原理，有助于掌握网络设备之间的通信机制，理解数据在网络中的传输过程。特别是对于网络工程和网络安全领域，从协议层面了解 ARP，有助于识别和防范诸如 ARP 欺骗等网络攻击，提高网络的安全防护能力。

通过本次实验，学员将亲自动手实现 ARP 协议的核心功能，包括 ARP 请求与响应的构建与解析、ARP 缓存表的管理等。这不仅加深了对 TCP/IP 协议栈的理解，也培养了实际编程和问题解决的能力。掌握 ARP 协议的实现，对后续学习更复杂的网络协议（如 IP、ICMP、TCP 和 UDP）以及从事网络相关工作都有重要的意义。

2 实验原理及方案

ARP（地址解析协议）是 TCP/IP 协议族中用于将 IP 地址解析为 MAC 地址的重要协议。IP 通信依赖于数据链路层的硬件地址（MAC 地址），而 ARP 负责动态地将网络层的 IP 地址转换为对应的数据链路层 MAC 地址，从而实现设备间的通信。ARP 协议的实现主要包括发送 ARP 请求、接收并处理 ARP 响应、更新 ARP 缓存、以及缓存超时机制。

2.1 ARP 的初始化

在一个典型的局域网中，设备通过 IP 地址进行网络层通信，但 IP 地址并不能直接用于数据链路层传输。以太网等数据链路层协议使用 MAC 地址进行通信，因此，发送设备需要将目标 IP 地址解析为 MAC 地址才能发送数据帧。

如果该设备的 ARP 缓存中没有目标设备的 MAC 地址映射，它会广播 ARP 请求，询问网络上哪个设备持有特定的 IP 地址。ARP 请求是一个以太网层的广播包，发送到子网内所有设备，只有持有目标 IP 地址的设备才会进行响应。

ARP 初始化的过程是设备发现并解析网络中其他设备的关键步骤。ARP 请求包含源设备的 IP 地址和 MAC 地址，而目标设备通过 ARP 响应提供其对应的 MAC 地址。这个机制确保设备能够通过网络层（IP 地址）和链路层（MAC 地址）之间建立正确的映射关系。

2.2 无回报 ARP 的生成

无回报 ARP（Gratuitous ARP），又称为“主动 ARP”或“自愿 ARP”，是一种特殊的 ARP 操作。与典型的 ARP 请求不同，无回报 ARP 并不是为了解析目标设备的 MAC 地址，而是设备主动向网络发送广播 ARP 包，通常用于更新网络中的 IP-MAC 映射关系、检测 IP 地址冲突等。

无回报 ARP 是设备主动广播自身的 IP 地址和 MAC 地址，不带有显式的 ARP 请求和响应互动。其主要目的是通知网络中其他设备更新其 ARP 缓存表中的信息。这种情况下，设备并不期待其他设备回应。它是单向广播的，通常被用于下列几种情况：

- 更新网络中的 ARP 表：当设备的 MAC 地址或 IP 地址发生变动时，可以主动发送无回报 ARP，以便通知网络中其他设备更新其 ARP 缓存。
- IP 冲突检测：设备在启动时，通过发送无回报 ARP 来检测是否有其他设备占用了相同的 IP 地址。如果另一台设备使用了相同的 IP 地址，它会回应此 ARP 广播，从而帮助设备检测到 IP 冲突。
- 负载均衡器和高可用性系统：当系统切换主备设备时，备设备通常会发送无回报 ARP 来通知网络中的所有节点其 IP-MAC 映射已经改变，避免继续向已下线的设备发送数据。

无回报 ARP 的生成过程如下：

1. 生成 ARP 广播包：设备在确定需要广播自身 IP-MAC 映射时，会生成一个 ARP 广播包。该包包含设备自身的 IP 地址和 MAC 地址，并且目标硬件地址设置为全 0，因为无回报 ARP 并不是请求对方设备的 MAC 地址，而是向网络中的所有设备广播自身的信息。
2. 设置操作码为 ARP 请求：尽管无回报 ARP 是主动广播，但它在帧结构中被标记为 ARP 请求（操作码为 1），这使得网络中的其他设备会将其视为一种信息广播，用于更新 ARP 缓存。
3. 发送广播：ARP 广播包通过以太网层进行传输，目标 MAC 地址为 FF:FF:FF:FF:FF:FF，即局域网内的所有设备都可以接收到此广播。

4. 网络中的设备处理：网络中所有收到此广播的设备会检查 ARP 包中的发送方 IP 地址和 MAC 地址，并将其更新到本地的 ARP 缓存表中。这样，即使该 IP 地址之前未出现在这些设备的 ARP 表中，它们也会记录并更新新的映射。

2.3 ARP 的输入处理

ARP（地址解析协议）的输入处理指的是设备在接收到 ARP 请求或响应时，如何对该 ARP 报文进行解析和处理，并据此更新设备的 ARP 缓存，或进一步采取必要的网络行为。ARP 输入处理的核心任务是解析报文，更新 ARP 缓存，并根据报文类型采取不同的操作。

在这部分有以下步骤：

- 接收 ARP 报文：设备通过网络接口接收到 ARP 报文，无论是广播还是单播形式。这些 ARP 报文可以是 ARP 请求、ARP 响应，或者是无回报 ARP。
- 解析 ARP 报文：设备对 ARP 报文进行解析，提取其中的关键信息。
- 检查报文有效性：设备检查 ARP 报文的有效性，包括检查硬件类型是否为以太网、协议类型是否为 IPv4、操作码是否为合法的请求或响应。如果报文不符合 ARP 协议规定，设备将丢弃该报文。
- 更新 ARP 缓存：根据 ARP 报文中的信息，设备更新自己的 ARP 缓存表。设备通常会把报文中的发送方 IP 地址和发送方 MAC 地址映射记录下来，以便将来进行快速的 IP 到 MAC 地址解析。
- 据操作码进行处理：不同类型的 ARP 报文有不同的处理方式：
  - 如果接收到的是 ARP 请求，设备需要检查目标 IP 地址是否与自身的 IP 地址匹配，如果匹配，则需要发送一个 ARP 响应包，告知请求设备自己的 MAC 地址。
  - 如果接收到的是 ARP 响应，设备会根据响应包中的信息，更新或添加到 ARP 缓存表，并不再发送进一步的响应。
  - 如果接收到的是无回报 ARP，设备会将报文中的 IP-MAC 映射记录下来，以更新其 ARP 缓存。

2.4 ARP 的超时重新请求机制

ARP（地址解析协议）的超时重新请求机制指的是设备在尝试解析某个 IP 地址到 MAC 地址时，若未能在设定的时间内收到响应，会采取的重发 ARP 请求的策略。这种机制旨在保证网络设备在通信中能够及时获取目标设备的 MAC 地址，并维持 ARP 缓存的准确性。

ARP 缓存存储的是 IP 地址与 MAC 地址之间的映射关系。在通信过程中，网络设备通常会先查询 ARP 缓存以查找目标设备的 MAC 地址。如果缓存中存在该 IP 地址的记录，设备会直接使用缓存中的 MAC 地址进行通信；如果没有找到相应记录，设备会发出 ARP 请求，广播请求目标 IP 地址对应的 MAC 地址。

如果设备在发送 ARP 请求后，未能在指定的时间内收到 ARP 响应，它会认为该 ARP 请求失败。这时，设备会重新发送 ARP 请求，通常会进行一定次数的重发，以确保能够成功解析目标设备的 MAC 地址。

3 实验环境

3.1 实验设备与软件

名称	型号或版本
物理机	联想 ThinkPad-Windows 10 22H4
虚拟机	Virtual Box-Windows 10 22H4
Wireshark	Wireshark 4.4.0
CMake	CMake 3.31.3

4 实验步骤

4.1 环境配置

4.1.1 虚拟机网络配置

安装 Windows 10 虚拟机，并配置物理机和虚拟机的 IP 地址，使其能够互相访问：

- 物理机网卡 IP 地址配置为 `192.168.254.1/24`；
- 虚拟机 IP 地址配置为 `192.168.254.3/24`。

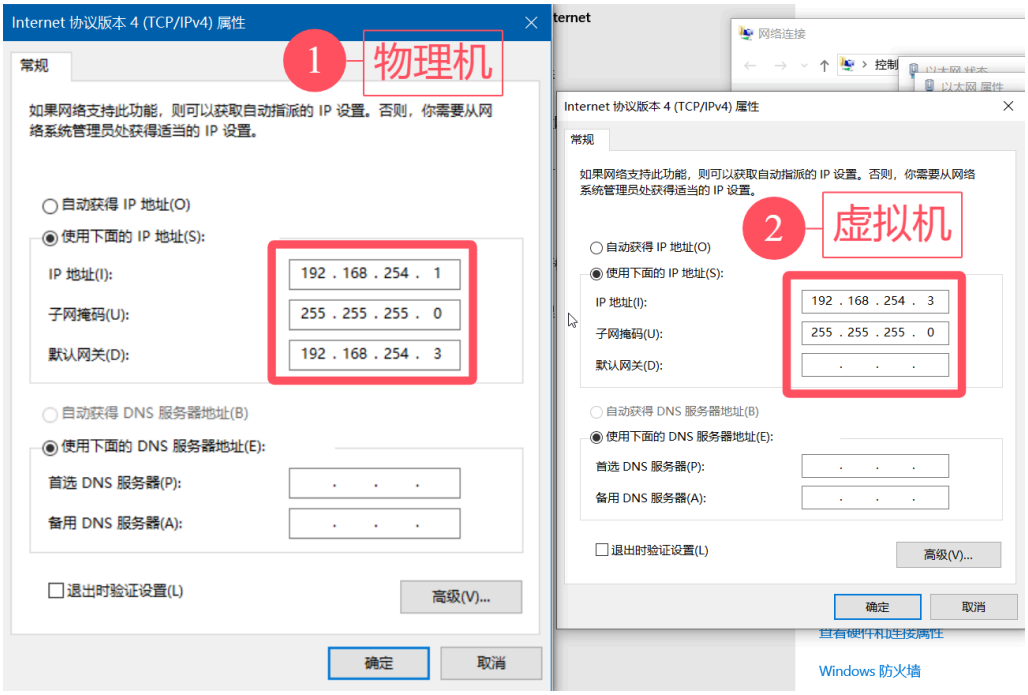

Figure 1: 物理机与虚拟机 IP 配置

配置好之后，在两边的命令行中分别使用 `ping` 命令测试是否能够互相访问。同时，在物理机上开启 Wireshark，以过滤条件 `icmp` 进行抓包，查看 IP 地址是否正确：

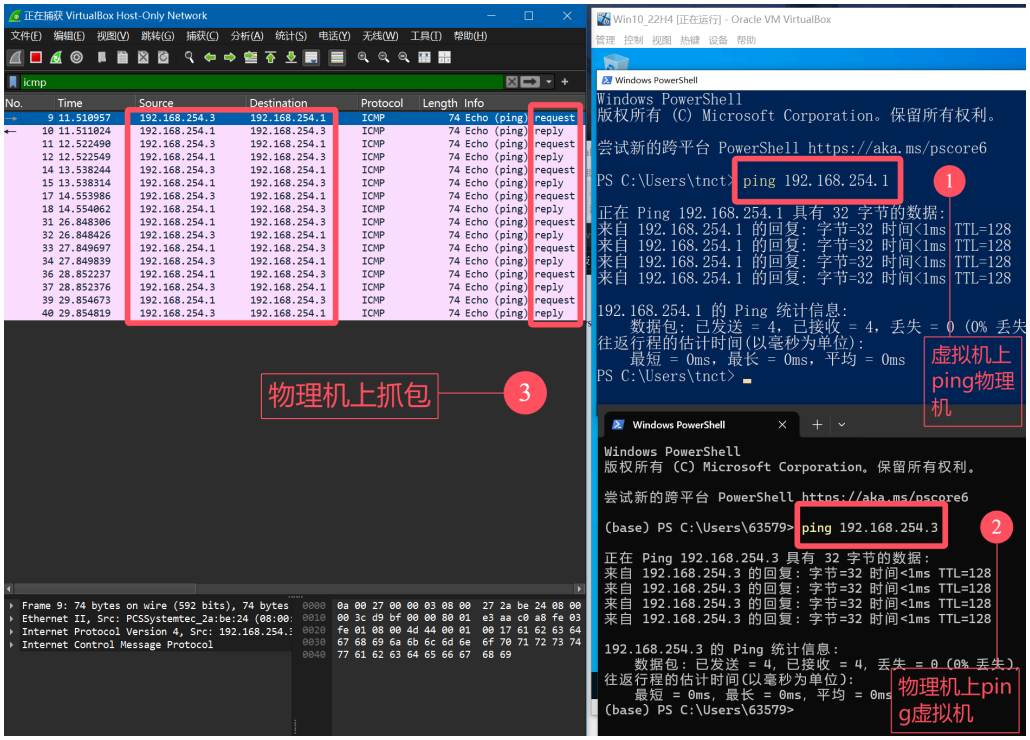

Figure 2: 环境配置 ping 通测试

从 Figure 2 中可以看到，物理机和虚拟机之间可以互相访问，且 Wireshark 抓包显示 IP 地址正确。

4.1.2 使用 CMake 运行项目

CMake 配置较为简单。首先，在开发工具中安装对应版本 CMake 插件。其次，在终端中进入项目根目录，在此使用`mkdir build`命令新建`build`文件夹并进入该文件夹。接下来，使用 CMake 工具生成对应的 Makefile 文件：`cmake -G"MinGW Makefiles" ..`。然后再运行`make`命令编译项目，最后使用`xnet.exe`命令即可运行项目：

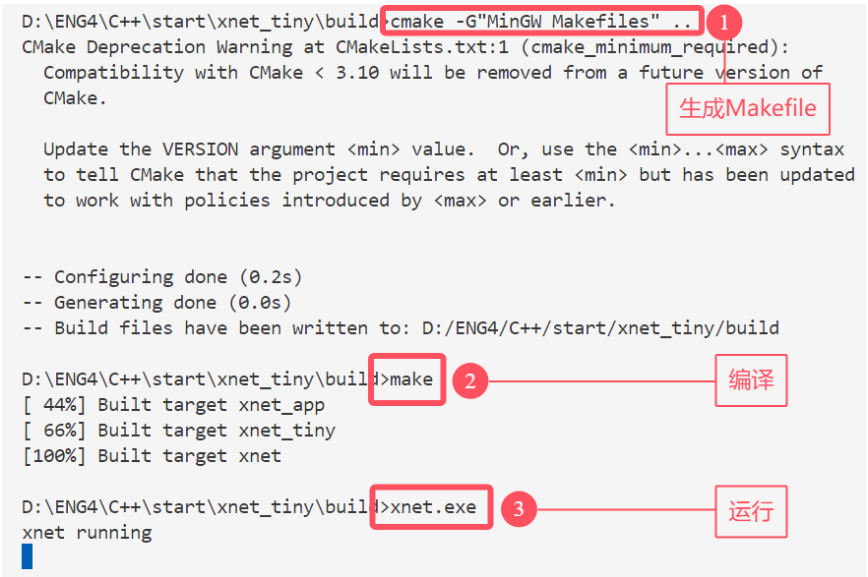

Figure 3: CMake 配置

其中，MinGW 是一个 Windows 下的 GNU 编译器套件，可以在 Windows 下编译出 Linux 下的可执行文件。`cmake -G"MinGW Makefiles" ..`命令的作用是配置使用 MinGW 编译器。

至此，环境配置结束。

4.2 实现 ARP 协议

代码已经实现了最基础的以太网协议，实现了以太网帧的封装和解封装。接下来在此基础上继续实现 ARP 协议。

4.2.1 相关数据结构

在`xnet_tiny.h`中定义 IP 地址长度以及数据结构：

```
#define XNET_IPV4_ADDR_SIZE       4         // IP 地址长度

// IP 地址
typedef union _xipaddr_t {
    uint8_t array[XNET_IPV4_ADDR_SIZE];     // 以数据形式存储的 ip
    uint32_t addr;                          // 32 位的 ip 地址
}xipaddr_t;
```

该数据结构定义了 IP 地址的数据结构，包括了 IP 地址的数组形式和 32 位的 IP 地址。

然后定义 MAC 地址的长度，以及 ARP 表项的结构体：

```
#define XNET_MAC_ADDR_SIZE        6       // MAC 地址长度

// ARP 表项
typedef struct _xarp_entry_t {
  xipaddr_t ipaddr;                       // ip 地址
  uint8_t macaddr[XNET_MAC_ADDR_SIZE];    // mac 地址
  uint8_t state;                          // 状态位
  uint16_t tmo;                           // 当前超时
  uint8_t retry_cnt;                      // 当前重试次数
}xarp_entry_t;
```

该结构体定义了 ARP 表项的数据结构，包括了 IP 地址、MAC 地址、状态位、超时时间和重试次数。

定义 ARP 表项的最大数量：

```
#define XARP_CFG_ENTRY_SIZE       8       // ARP 表大小
```

随后，在`xnet_tiny.c`中，将 ARP 表定义为全局变量，并定义一个表项指针，方便后续代码编写：

```
static xarp_entry_t arp_table[XARP_CFG_ENTRY_SIZE]; // ARP 表
static xarp_entry_t* arp_entry;                     // ARP 表项指针
```

4.2.2 ARP 表初始化

接下来编写 ARP 表的初始化函数。首先在`xnet_tiny.h`中定义 ARP 表项的第一个状态：

```
#define XARP_ENTRY_FREE          0 // ARP 表项空闲
```

然后在`xnet_tiny.c`中定义初始化函数`void xarp_init(void)`：

```
// ARP 初始化
void xarp_init(void) {
    for (arp_entry = arp_table;
         arp_entry < XARP_CFG_ENTRY_SIZE * sizeof(xarp_entry_t) + arp_table;
         arp_entry = arp_entry + sizeof(xarp_entry_t))
    {
        arp_entry->state = XARP_ENTRY_FREE; // 此处用到了上面定义的状态
    }
    arp_entry = arp_table;
}
```

初始化函数`void xarp_init(void)`是一个循环。首先将前面定义的全局表项指针指向 ARP 表的第一个表项，循环结束条件为指针指向的表项的地址超过 ARP 表的最后一个表项的地址。循环会遍历 ARP 表中的所有表项，将表项状态初始化为`XARP_STATE_FREE`。最后，函数会将表项指针指向第一个表项，避免其他初始化过程中可能的指针越界问题。

最后，在协议栈的初始化函数中添加`xarp_init()`：

```
void xnet_init (void) {
  ethernet_init();  // 初始化以太网
  xarp_init();      // *初始化 ARP
}
```

4.2.3 ARP 报文

接下来编写无回报 ARP 报文的相关函数，所以需要先定义 ARP 报文结构，以及它所用到的相关结构。

首先在`xnet_tiny.h`中定义 ARP 报文中的几个字段。可以靠 Wireshark 抓包分析来获取这些字段的值，下面是一个示例，展示通过抓包来获取`XNET_PROTOCOL_IP = 0x0800`：

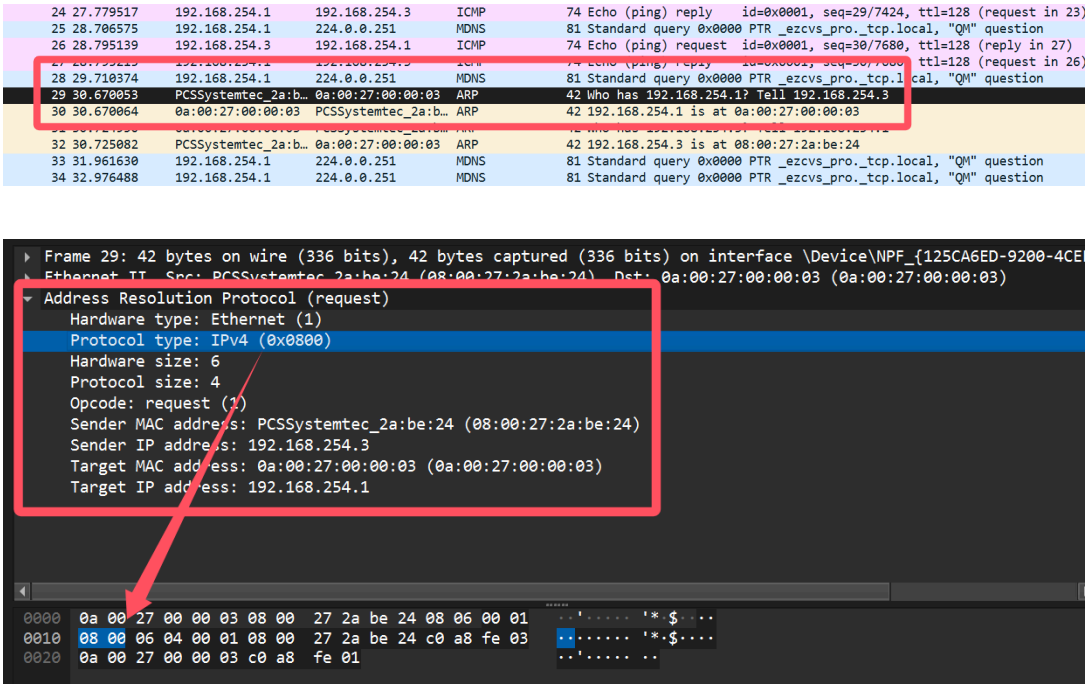

Figure 4: 通过抓包分析来获取字段值

代码编写如下：

```
#define XARP_HW_ETHER                0x1          // 硬件类型：以太网
#define XARP_REQUEST                 0x1          // Opcode：ARP 请求包
#define XARP_REPLY                   0x2          // Opcode：ARP 响应包

typedef enum _xnet_protocol_t {
    XNET_PROTOCOL_ARP = 0x0806,      // ARP 协议
    XNET_PROTOCOL_IP = 0x0800,       // IPv4 协议
    XNET_PROTOCOL_ICMP = 1,          // ICMP 协议
}xnet_protocol_t;
```

然后定义 ARP 报文的数据结构：

```
typedef struct _xarp_packet_t {
    uint16_t hw_type, pro_type;                 // 硬件类型和协议类型
    uint8_t hw_len, pro_len;                    // 硬件地址长 + 协议地址长
    uint16_t opcode;                            // 请求/响应
    uint8_t sender_mac[XNET_MAC_ADDR_SIZE];     // 发送包硬件地址
    uint8_t sender_ip[XNET_IPV4_ADDR_SIZE];     // 发送包协议地址
    uint8_t target_mac[XNET_MAC_ADDR_SIZE];     // 接收方硬件地址
    uint8_t target_ip[XNET_IPV4_ADDR_SIZE];     // 接收方协议地址
}xarp_packet_t;
```

然后，使用前面定义的 `union xipaddr_t` 结构，在 `xnet_tiny.h` 和 `xnet_tiny.c` 中定义 ARP 报文发送函数需要用到的 IP 地址、组播 MAC 地址：

```
// xnet_tiny.h
#define XNET_CFG_NETIF_IP               {192, 168, 254, 2}  // 本项目模拟出的网卡的 IP

// xnet_tiny.c
static const xipaddr_t netif_ipaddr = XNET_CFG_NETIF_IP;
static const uint8_t ether_broadcast[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
```

至此，定义 ARP 报文的数据结构结束。

4.2.4 ARP 报文发送函数

下面编写 ARP 报文发送函数`xarp_make_request(const xipaddr_t * ipaddr)`。

```
/**
 * 产生一个 ARP 请求，请求网络指定 ip 地址的机器发回一个 ARP 响应
 * @param ipaddr 请求的 IP 地址
 * @return 请求结果
 */
xnet_err_t xarp_make_request(const xipaddr_t * ipaddr) {
    xarp_packet_t* arp_packet;
    xnet_packet_t * packet = xnet_alloc_for_send(sizeof(xarp_packet_t));

    arp_packet = (xarp_packet_t *)packet->data;
    arp_packet->hw_type = swap_order16(XARP_HW_ETHER); // 设置硬件类型为以太网
    arp_packet->pro_type = swap_order16(XNET_PROTOCOL_IP); // 设置协议类型为 IP
    arp_packet->hw_len = XNET_MAC_ADDR_SIZE; // 设置硬件地址长度
    arp_packet->pro_len = XNET_IPV4_ADDR_SIZE; // 设置协议地址长度
    arp_packet->opcode = swap_order16(XARP_REQUEST); // 设置操作码为 ARP 请求
    // 复制发送方 MAC 地址
    memcpy(arp_packet->sender_mac, netif_mac, XNET_MAC_ADDR_SIZE);
    // 复制发送方 IP 地址
    memcpy(arp_packet->sender_ip, netif_ipaddr.array, XNET_IPV4_ADDR_SIZE);
    // 目标 MAC 地址清零
    memset(arp_packet->target_mac, 0, XNET_MAC_ADDR_SIZE);
    // 复制目标 IP 地址
    memcpy(arp_packet->target_ip, ipaddr->array, XNET_IPV4_ADDR_SIZE);
    // 通过以太网发送 ARP 请求
    return ethernet_out_to(XNET_PROTOCOL_ARP, ether_broadcast, packet);
}
```

这个函数的主要功能是生成并发送一个 ARP 请求报文，以请求指定 IP 地址（即函数的输入`const xipaddr_t * ipaddr`）的机器返回其 MAC 地址。函数的具体步骤如下：

1. 分配一个用于发送的 ARP 数据包`arp_packet`，并将其数据段设置为 ARP 报文结构。
2. 设置 ARP 报文的各个字段，包括硬件类型`hw_type`、协议类型`pro_type`、硬件地址长度`hw_len`、协议地址长度`pro_len`、操作码`opcode`（设置为`XARP_REQUEST`）等。
3. 复制发送方（即本项目模拟出的网卡）的 MAC 地址和 IP 地址到 ARP 报文中。
4. 将目标 MAC 地址字段清零，并复制目标 IP 地址到 ARP 报文中。
5. 最后，通过以太网发送该 ARP 请求报文，返回发送结果（状态码）。

4.2.5 启动时的 ARP 请求

在以太网协议的初始化函数`static xnet_err_t ethernet_init(void)`中添加一个 ARP 请求：

```
/**
 * 以太网初始化
 * @return 初始化结果
 */
static xnet_err_t ethernet_init (void) {
    xnet_err_t err = xnet_driver_open(netif_mac);
    if (err < 0) return err;

    return xarp_make_request(&netif_ipaddr); // 发送 ARP 请求
}
```

这样，当协议栈初始化时，会发送一个 ARP 请求。

下面用 Wireshark 抓包来验证 ARP 请求是否发送成功。首先，重新编译项目；其次，开启 Wireshark 抓包；最后，启动程序：

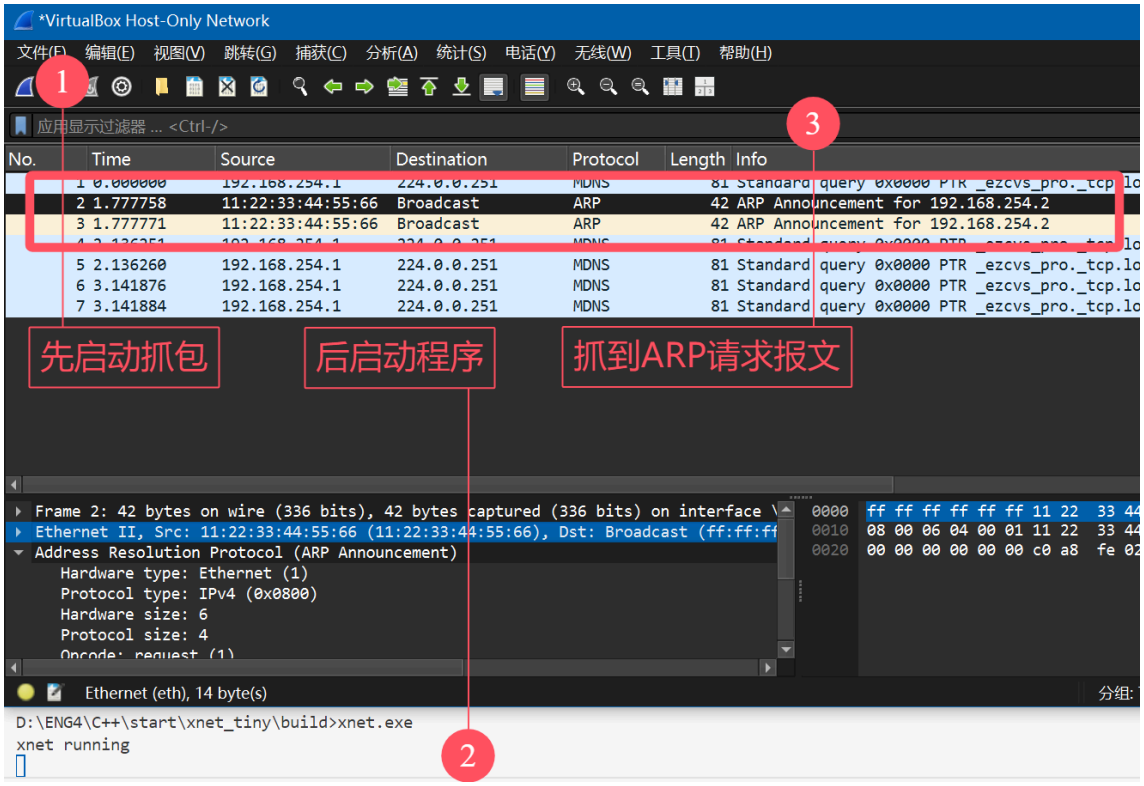

Figure 7: 启动时的 ARP 请求

从 Figure 7 中可以看到，ARP 请求发送成功，说明编写至此的代码没有问题。

4.2.6 ARP 报文接收函数

ARP 报文接收函数主要功能是处理接收到的 ARP 报文，包括解析报文、更新 ARP 表、发送 ARP 响应等。下面，根据这些需求编写 ARP 报文接收函数`void xarp_in(xnet_packet_t * packet)`：

```c
/**
 * 处理接收到的 ARP 包
 * @param packet 输入的 ARP 包
 */
void xarp_in(xnet_packet_t * packet) {
    // 检查包的大小是否符合 ARP 包的最小长度要求
    if (packet->size >= sizeof(xarp_packet_t)) {
        xarp_packet_t * arp_packet = (xarp_packet_t *) packet->data;
        uint16_t opcode = swap_order16(arp_packet->opcode);

        // 检查包的合法性，包括硬件类型、硬件地址长度、协议类型、协议地址长度和操作码
        if ((swap_order16(arp_packet->hw_type) != XARP_HW_ETHER) ||
            (arp_packet->hw_len != XNET_MAC_ADDR_SIZE) ||
            (swap_order16(arp_packet->pro_type) != XNET_PROTOCOL_IP) ||
            (arp_packet->pro_len != XNET_IPV4_ADDR_SIZE)
            || ((opcode != XARP_REQUEST) && (opcode != XARP_REPLY))) {
            return;
        }

        // 只处理目标 IP 地址为自己的 ARP 请求或响应包
        if (!xipaddr_is_equal_buf(&netif_ipaddr, arp_packet->target_ip)) {
            return;
        }

        // 根据操作码进行处理
        switch (swap_order16(arp_packet->opcode)) {
            case XARP_REQUEST:  // 处理 ARP 请求，发送 ARP 响应并更新 ARP 表项
                xarp_make_response(arp_packet);
                update_arp_entry(arp_packet->sender_ip, arp_packet->sender_mac);
                break;
            case XARP_REPLY:    // 处理 ARP 响应，更新 ARP 表项
                update_arp_entry(arp_packet->sender_ip, arp_packet->sender_mac);
                break;
        }
    }
}
```

该函数主要功能是处理接收到的 ARP 包。首先进行简单的长度判断，避免后续字段读取失败造成内存错误。随后检查包的合法性，包括硬件类型、硬件地址长度、协议类型、协议地址长度和操作码。APR 响应只要求机器处理目标 IP 地址为自己的 ARP 请求或响应包，所以使用 `if (!xipaddr_is_equal_buf(&netif_ipaddr, arp_packet->target_ip))` 来判断。最后，根据操作码进行处理，分别处理 ARP 请求和 ARP 响应：

- ARP 请求：发送 ARP 响应（`xarp_make_response(...)`）并更新 ARP 表项（`update_arp_entry(...)`）；
- ARP 响应：只需要更新 ARP 表项。

其中，用到的宏 `xipaddr_is_equal_buf()` 函数用于比较两个 IP 地址是否相等，实现如下：

```c
// 比较 IP 地址是否相等
#define xipaddr_is_equal_buf(addr, buf)   (memcmp(
                                            (addr)->array,
                                            (buf),
                                            XNET_IPV4_ADDR_SIZE
                                            )
```

```
                                        == 0
                                    )
```

然后，需要编写上面函数中调用的两个函数：`xarp_make_response()`和`update_arp_entry()`。

`xarp_make_response()`函数主要功能是：输入一个 ARP 请求包，通过此包内的源信息，生成对应的 ARP 响应，并发送出去。具体代码如下：

```
/**
 * 生成一个 ARP 响应
 * @param arp_packet 接收到的 ARP 请求包
 * @return 生成结果
 */
xnet_err_t xarp_make_response(xarp_packet_t * arp_packet) {
    xarp_packet_t* response_packet;
    xnet_packet_t * packet = xnet_alloc_for_send(sizeof(xarp_packet_t));

    response_packet = (xarp_packet_t *)packet->data;
    response_packet->hw_type = swap_order16(XARP_HW_ETHER); // 设置硬件类型为以太网
    response_packet->pro_type = swap_order16(XNET_PROTOCOL_IP); // 设置协议类型为 IP
    response_packet->hw_len = XNET_MAC_ADDR_SIZE; // 设置硬件地址长度
    response_packet->pro_len = XNET_IPV4_ADDR_SIZE; // 设置协议地址长度
    response_packet->opcode = swap_order16(XARP_REPLY); // 设置操作码为 ARP 响应
    // 复制目标 MAC 地址
    memcpy(response_packet->target_mac, arp_packet->sender_mac, XNET_MAC_ADDR_SIZE);
    // 复制目标 IP 地址
    memcpy(response_packet->target_ip, arp_packet->sender_ip, XNET_IPV4_ADDR_SIZE);
    // 复制发送方 MAC 地址
    memcpy(response_packet->sender_mac, netif_mac, XNET_MAC_ADDR_SIZE);
    // 复制发送方 IP 地址
    memcpy(response_packet->sender_ip, netif_ipaddr.array, XNET_IPV4_ADDR_SIZE);
    // 通过以太网发送 ARP 响应
    return ethernet_out_to(XNET_PROTOCOL_ARP, ether_broadcast, packet);
}
```

可以发现此函数与前面的 ARP 请求函数`xarp_make_request()`非常相似，只是操作码不同，此处为`XARP_REPLY`，其他字段均从源 ARP 请求报文中获取，并填入对应区域。

`update_arp_entry()`函数主要功能是更新所有 ARP 表项，附带一定的可视化功能。具体代码如下：

```
/**
 * 更新 ARP 表项
 * @param src_ip 源 IP 地址
 * @param mac_addr 对应的 mac 地址
 */
static void update_arp_entry(uint8_t* src_ip, uint8_t* mac_addr) {
    for (arp_entry = arp_table;
         arp_entry < XARP_CFG_ENTRY_SIZE * sizeof(xarp_entry_t) + arp_table;
         arp_entry = arp_entry + sizeof(xarp_entry_t))
    {
        // 检查 ARP 表项是否为空或者是否与给定的源 IP 地址匹配且状态不是有效的
        if (arp_entry->state == XARP_ENTRY_FREE ||
```

```
        ( arp_entry->state == XARP_ENTRY_OK
          && xipaddr_is_equal_buf(&arp_entry->ipaddr, src_ip)
        ))
    {
        // 更新 ARP 表项中的 IP 地址和 MAC 地址
        memcpy(arp_entry->ipaddr.array, src_ip, XNET_IPV4_ADDR_SIZE);
        memcpy(arp_entry->macaddr, mac_addr, 6);
        printf("learned👍🧐mac addr: \n");
        for (
              int i = 0;
              i < sizeof(mac_addr) / sizeof(mac_addr[0]);
              ++i)
        {
            printf("%02X%c",
                   mac_addr[i],
                   i < sizeof(mac_addr) / sizeof(mac_addr[0]) - 1 ? ':' : '\n'
                  );
        }
        // 设置 ARP 表项状态为有效
        arp_entry->state = XARP_ENTRY_OK;
        // 设置 ARP 表项的超时时间
        arp_entry->tmo = XARP_CFG_ENTRY_OK_TMO;
        // 设置 ARP 表项的重试次数
        arp_entry->retry_cnt = XARP_CFG_MAX_RETRIES;
        print_arp_table(); // 打印完整的 ARP 表
        return; // 更新后退出函数
    }
}

// 如果 ARP 表已满，采用 LRU 策略替换最老的表项
arp_entry = arp_table; // 重置 arp_entry 指向表头
xarp_entry_t* oldest_entry = NULL;
uint32_t oldest_tmo = 0xFFFFFFFF;
for (arp_entry = arp_table;
     arp_entry < XARP_CFG_ENTRY_SIZE * sizeof(xarp_entry_t) + arp_table;
     arp_entry = arp_entry + sizeof(xarp_entry_t))
{
    if (arp_entry->tmo < oldest_tmo) {
        oldest_tmo = arp_entry->tmo;
        oldest_entry = arp_entry;
    }
}
if (oldest_entry != NULL) {
    // 更新最老的 ARP 表项
    memcpy(oldest_entry->ipaddr.array, src_ip, XNET_IPV4_ADDR_SIZE);
    memcpy(oldest_entry->macaddr, mac_addr, 6);
    printf("learned👍🧐mac addr: \n");
    for (int i = 0; i < sizeof(mac_addr) / sizeof(mac_addr[0]); ++i){
        printf("%02X%c", mac_addr[i],
               i < sizeof(mac_addr) / sizeof(mac_addr[0]) - 1 ? ':' : '\n');
    }
    // 设置 ARP 表项状态为有效
    oldest_entry->state = XARP_ENTRY_OK;
    // 设置 ARP 表项的超时时间
    oldest_entry->tmo = XARP_CFG_ENTRY_OK_TMO;
    // 设置 ARP 表项的重试次数
    oldest_entry->retry_cnt = XARP_CFG_MAX_RETRIES;
    print_arp_table(); // 打印完整的 ARP 表
```

```
    }
}
```

这个函数很长。它主要功能是更新 ARP 表项。更新分为两种情况：

- ARP 表还有空闲表项
- ARP 表已满，采用 LRU 策略替换最老的表项

首先，函数通过遍历 ARP 表中的所有表项，检查表项是否为空或者是否与给定的源 IP 地址匹配且状态不是有效的。如果满足条件，则更新 ARP 表项中的 IP 地址和 MAC 地址，并设置表项状态为有效，设置超时时间和重试次数（最后，会打印完整的 ARP 表）。如果 ARP 表已满，则采用 LRU 策略替换最老的表项。函数会遍历 ARP 表，找到超时时间最小的表项，并更新该表项的 IP 地址和 MAC 地址，设置表项状态为有效，设置超时时间和重试次数，最后打印完整的 ARP 表。

用到的打印函数实现如下：

```
/**
 * 打印完整的 ARP 表
 */
void print_arp_table() {
    printf("\n----ARP Table----\n");
    for (arp_entry = arp_table;
         arp_entry < XARP_CFG_ENTRY_SIZE * sizeof(xarp_entry_t) + arp_table;
         arp_entry = arp_entry + sizeof(xarp_entry_t))
    {
        if (arp_entry->state != XARP_ENTRY_FREE) {
            printf("IP: ");
            for (int i = 0; i < XNET_IPV4_ADDR_SIZE; ++i) {
                printf("%d%c",
                       arp_entry->ipaddr.array[i],
                       i < XNET_IPV4_ADDR_SIZE - 1 ? '.' : '\n'
                      );
            }
            printf("MAC: ");
            for (int i = 0; i < 6; ++i) {
                printf("%02X%c", arp_entry->macaddr[i], i < 5 ? ':' : '\n');
            }
            printf(
                   "State: %s\n",
                   arp_entry->state == XARP_ENTRY_FREE ? "FREE" :
                   arp_entry->state == XARP_ENTRY_RESOLVING ? "RESOLVING" : "OK"
                  );
        }
    }
    printf("\n-----------------\n");
}
```

最后，需要在以太网帧接收函数中添加 ARP 报文的处理：

```
/**
 * 以太网数据帧输入输出
 * @param packet 待处理的包
 */
```

```c
static void ethernet_in (xnet_packet_t * packet) {
    // 至少要比头部数据大
    if (packet->size <= sizeof(xether_hdr_t)) {
        return;
    }
    // 根据协议类型分发到不同的处理函数
    xether_hdr_t* hdr = (xether_hdr_t*)packet->data;
    switch (swap_order16(hdr->protocol)) {
        case XNET_PROTOCOL_ARP:
            // 移除以太网头部，处理 ARP 协议
            remove_header(packet, sizeof(xether_hdr_t));
            xarp_in(packet);
            break;
        case XNET_PROTOCOL_IP: {
            break;
        }
    }
}
```

其中，主要在`case XNET_PROTOCOL_ARP`中添加了对 ARP 报文的处理。

在继续之前，再次使用 Wireshark 检验这部分代码编写。重新编译后，按照以下流程进行检验：

- 开启 Wireshark 抓包；
- 运行本程序；
- 在虚拟机上 ping 本程序，以触发 ARP 请求；
- 查看 Wireshark 抓包结果和程序输出。

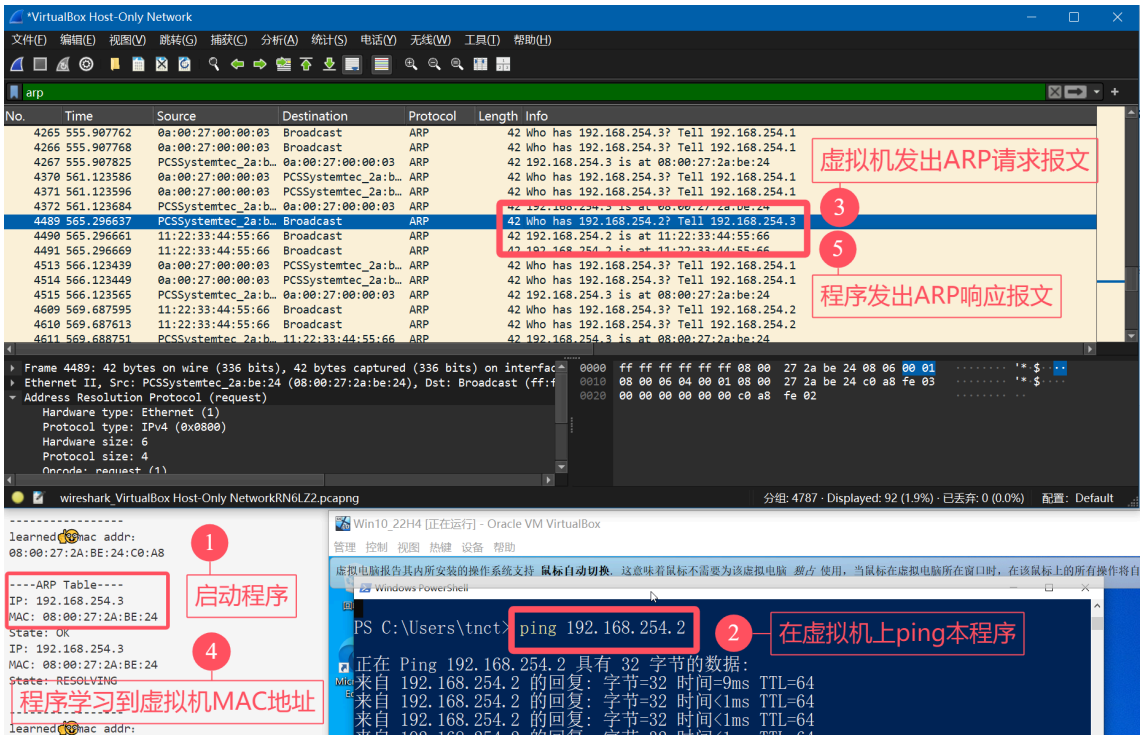

Figure 8: ARP 请求响应

从 Figure 8 中可以看到，ARP 响应都发送成功，程序输出中也表明学习到了虚拟机的 MAC 地址，说明代码编写正确。

4.2.7 ARP 超时重传

首先，需要定义 ARP 表项的其他两种状态[1]：解析成功、和正在解析（即已发出重传的 ARP 请求报文，但还未收到响应）：

```
#define XARP_ENTRY_OK              1       // ARP 表项解析成功
#define XARP_ENTRY_RESOLVING       2       // ARP 表项正在解析
#define XARP_TIMER_PERIOD          1       // ARP 扫描周期，1s 足够
```

然后需要定义超时时间和重试次数：

```
#define XARP_CFG_ENTRY_OK_TMO       (10)  // ARP 表项超时时间
#define XARP_CFG_ENTRY_PENDING_TMO  (2)   // ARP 表项挂起超时时间
#define XARP_CFG_MAX_RETRIES         4    // ARP 表挂起时重试查询次数
```

在 `xnet_tiny.c` 中，`xarp_poll` 函数负责定期检查 ARP 表项的状态，并根据需要触发重传。具体实现如下：

```
/**
 * 查询 ARP 表项是否超时，超时则重新请求
 */
void xarp_poll(void) {
    // 检查 ARP 定时器是否超时
    if (xnet_check_tmo(&arp_timer, XARP_TIMER_PERIOD)) {
        for ( arp_entry = arp_table;
              arp_entry < XARP_CFG_ENTRY_SIZE * sizeof(xarp_entry_t) + arp_table;
              arp_entry = arp_entry + sizeof(xarp_entry_t))
        {
            switch (arp_entry->state) {
            case XARP_ENTRY_RESOLVING:
                // 如果 ARP 表项正在解析中，检查超时计数器
                if (--arp_entry->tmo == 0) {
                    // 如果重试次数用完，释放 ARP 表项
                    if (arp_entry->retry_cnt-- == 0) {
                        arp_entry->state = XARP_ENTRY_FREE;
                    }
                    else {
                        // 否则继续重试，发送 ARP 请求
                        xarp_make_request(&arp_entry->ipaddr);
                        arp_entry->state = XARP_ENTRY_RESOLVING;
                        arp_entry->tmo = XARP_CFG_ENTRY_PENDING_TMO;
                    }
                }
                break;
            case XARP_ENTRY_OK:
                // 如果 ARP 表项有效，检查超时计数器
                if (--arp_entry->tmo == 0) {
                    // 超时后重新发送 ARP 请求
```

[1] 第一种状态已经在 Section 4.2.2 中定义过了。

```
                    xarp_make_request(&arp_entry->ipaddr);
                    arp_entry->state = XARP_ENTRY_RESOLVING;
                    arp_entry->tmo = XARP_CFG_ENTRY_PENDING_TMO;
                }
                break;
            }
        }
    }
}
```

该函数主要功能是定时检查 ARP 表项的状态，具体步骤如下：

1. 定时检查：`xarp_poll()` 函数会定期检查 ARP 表项的状态，检查周期由 `XARP_TIMER_PERIOD` 定义（1 秒）。
2. 状态判断：
   - 如果表项状态为 `XARP_ENTRY_RESOLVING`（正在解析中），则检查超时计数器。如果超时且重试次数用完，则释放该表项；否则，重新发送 ARP 请求并重置超时计数器。
   - 如果表项状态为 `XARP_ENTRY_OK`（有效），则检查超时计数器。如果超时，则重新发送 ARP 请求并将表项状态设置为 `XARP_ENTRY_RESOLVING`。
3. 重传 ARP 请求：通过调用 `xarp_make_request` 函数，重新发送 ARP 请求以获取目标 IP 地址对应的 MAC 地址。

下面是用 Wireshark 抓包验证 ARP 超时重传的结果。重新编译后，直接运行程序，在虚拟机上 ping 本程序，以触发 ARP 请求。本程序学习完成之后，每秒钟会发送一次 ARP 请求。Wireshark 抓包结果如下：

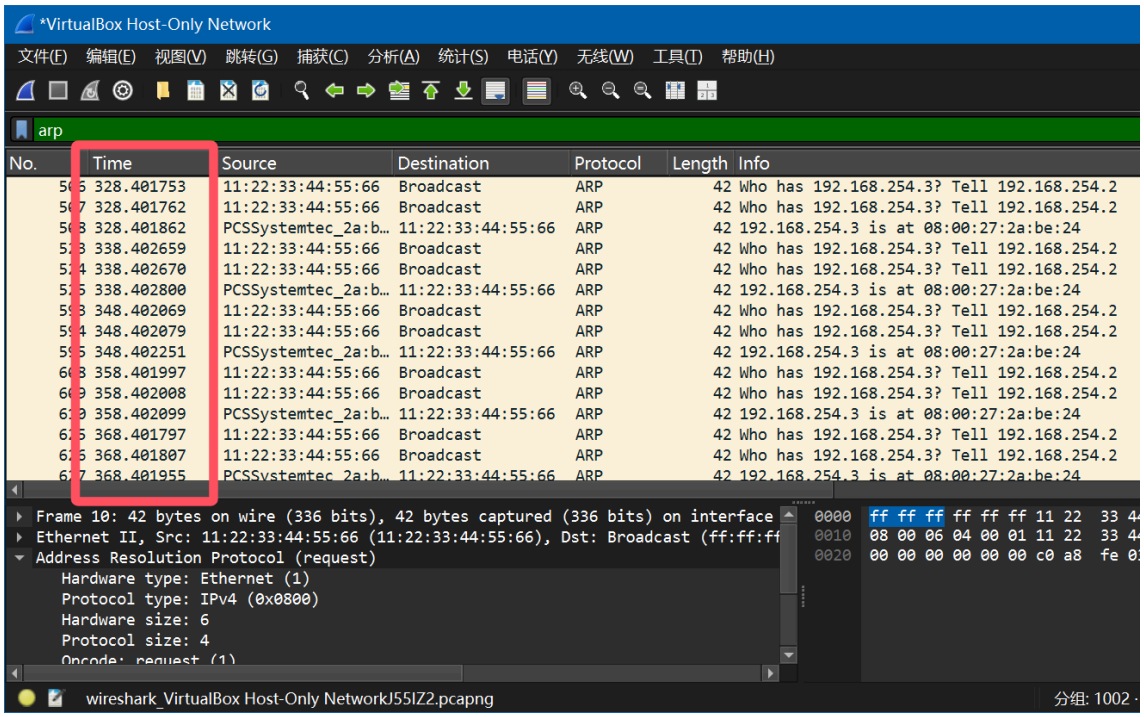

Figure 9: 固定间隔的 ARP 超时重传

注意 Figure 9 中左侧的 `Time` 一列，从上到下程序发出的 ARP 请求的时间依次增加 10 秒。这表明，ARP 请求每 10 秒钟发送一次，在上面代码中的这一行定义过的 ARP 表项超时时间生效，ARP 请求重传成功。

```
#define XARP_CFG_ENTRY_OK_TMO       (10)  // ARP 表项超时时间
```

至此，ARP 协议的实现完成。

4.3 实现 IP 协议

以太网之上，除了 ARP 协议，还有 IP 协议。IP 协议是网络层协议，负责将数据包从源主机传输到目的主机。IP 协议的数据包称为 IP 数据报，包含了源 IP 地址和目的 IP 地址。

与前面通过抓包来获取 ARP 数据包的各字段值的方法类似，IP 数据包各字段值也可以通过抓包获取，但更方便的做法是查询 RFC 文档[2]来获取，此处不再展示。下面来实现 IP 协议。

4.3.1 定义 IP 数据报

在 `xnet_tiny.h` 中定义 IP 数据报的结构体：

```
typedef struct _xip_hdr_t {
    uint8_t hdr_len : 4;                // 首部长，4 字节为单位
    uint8_t version : 4;                // 版本号
    uint8_t tos;                        // 服务类型
    uint16_t total_len;                 // 总长度
    uint16_t id;                        // 标识符
    uint16_t flags_fragment;            // 标志与分段
    uint8_t ttl;                        // 存活时间
    uint8_t protocol;                   // 上层协议
    uint16_t hdr_checksum;              // 首部校验和
    uint8_t src_ip[XNET_IPV4_ADDR_SIZE]; // 源 IP
    uint8_t dest_ip[XNET_IPV4_ADDR_SIZE]; // 目标 IP
} xip_hdr_t;
```

该结构体定义了 IP 数据报的各个字段，包括：

- `hdr_len`：IP 头部的长度，以 4 字节为单位。
- `version`：IP 版本号，通常为 IPv4（值为 4）。
- `tos`：服务类型，用于指定数据报的优先级。
- `total_len`：IP 数据报的总长度。
- `id`：标识符，用于标识 IP 数据报。
- `flags_fragment`：标志和分段信息，用于 IP 分片。
- `ttl`：生存时间，用于防止数据报在网络中无限循环。
- `protocol`：上层协议类型，如 ICMP、TCP 或 UDP。
- `hdr_checksum`：IP 头部的校验和，用于检测数据报是否损坏。
- `src_ip` 和 `dest_ip`：源 IP 地址和目标 IP 地址。

[2] 文档地址：RFC 791: Internet Protocol

4.3.2 实现 IP 输入

IP 输入函数 `xip_in` 负责处理接收到的 IP 数据报。在 `xnet_tiny.c` 中实现该函数：

```
void xip_in(xnet_packet_t * packet) {
    xip_hdr_t* iphdr = (xip_hdr_t*)packet->data;
    uint32_t total_size, header_size;
    uint16_t pre_checksum;
    xipaddr_t src_ip;

    // 检查 IP 版本号是否为 IPv4
    if (iphdr->version != XNET_VERSION_IPV4) {
        return;
    }

    // 检查头部长度和总长度是否符合要求
    header_size = iphdr->hdr_len * 4;
    total_size = swap_order16(iphdr->total_len);
    if (
      (header_size < sizeof(xip_hdr_t))
      || ((total_size < header_size)
      || (packet->size < total_size))
      )
      return;


    // 校验头部的校验和是否正确
    pre_checksum = iphdr->hdr_checksum;
    iphdr->hdr_checksum = 0;
    if (pre_checksum != checksum16((uint16_t*)iphdr, header_size, 0, 1)) {
        return;
    }

    // 检查目标 IP 地址是否为本机 IP
    if (!xipaddr_is_equal_buf(&netif_ipaddr, iphdr->dest_ip)) {
        return;
    }

    // 根据协议类型分发到不同的处理函数
    xipaddr_from_buf(&src_ip, iphdr->src_ip);
    switch(iphdr->protocol) {
        case XNET_PROTOCOL_ICMP:
            remove_header(packet, header_size);
            xicmp_in(&src_ip, packet);
            break;
        default:
            break;
    }
}
```

函数逻辑、功能比较简单，概括如下：检查 IP 版本号是否为 IPv4。然后验证 IP 头部的长度和总长度是否符合要求。再校验 IP 头部的校验和是否正确、检查目标 IP 地址是否为本机 IP。最后根据上层协议类型（如 ICMP），将数据报分发到相应的处理函数。

4.3.3 实现 IP 输出

IP 输出函数 `xip_out` 负责发送 IP 数据报。在 `xnet_tiny.c` 中，我们实现了该函数：

```
xnet_err_t xip_out(xnet_protocol_t protocol, xipaddr_t* dest_ip, xnet_packet_t * packet)
{
    static uint32_t ip_packet_id = 0; // 静态变量，用于生成唯一的 IP 包 ID
    xip_hdr_t * iphdr;

    add_header(packet, sizeof(xip_hdr_t)); // 添加 IP 头部
    iphdr = (xip_hdr_t*)packet->data; // 获取 IP 头部指针
    iphdr->version = XNET_VERSION_IPV4; // 设置 IP 版本号为 IPv4
    iphdr->hdr_len = sizeof(xip_hdr_t) / 4; // 设置 IP 头部长度
    iphdr->tos = 0; // 设置服务类型
    iphdr->total_len = swap_order16(packet->size); // 设置总长度
    iphdr->id = swap_order16(ip_packet_id); // 设置包 ID
    iphdr->flags_fragment = 0; // 设置标志和片偏移
    iphdr->ttl = XNET_IP_DEFAULT_TTL; // 设置生存时间
    iphdr->protocol = protocol; // 设置上层协议类型
    memcpy(iphdr->dest_ip, dest_ip->array, XNET_IPV4_ADDR_SIZE); // 设置目标 IP 地址
    // 设置源 IP 地址
    memcpy(iphdr->src_ip, netif_ipaddr.array, XNET_IPV4_ADDR_SIZE);
    iphdr->hdr_checksum = 0; // 初始化校验和字段
    // 计算并设置校验和
    iphdr->hdr_checksum = checksum16((uint16_t *)iphdr, sizeof(xip_hdr_t), 0, 1);

    ip_packet_id++; // 增加包 ID
    return ethernet_out(dest_ip, packet); // 通过以太网发送 IP 包
}
```

该函数的主要功能是：

1. 添加 IP 头部到数据报中。
2. 设置 IP 头部的各个字段，包括版本号、头部长度、总长度、包 ID、生存时间、上层协议类型、源 IP 地址和目标 IP 地址。
3. 计算并设置 IP 头部的校验和。
4. 通过以太网发送 IP 数据报。

4.4 实现 ICMP 协议

实现了 ARP 和 IP 协议之后，下面来实现 ICMP 协议。ICMP 协议是网络层协议，用于在 IP 网络中传递控制消息。ICMP 数据报的数据部分包含了 ICMP 报文的类型、代码和校验和等字段。实现 ICMP 的主要目的在于实现 ping 功能，即实现 ICMP 的 ping 响应。

4.4.1 定义 ICMP 数据报

ICMP（Internet Control Message Protocol）是网络层协议，用于在 IP 网络中传递控制消息。在 `xnet_tiny.h` 中，我们定义了 ICMP 数据报的结构体：

```
typedef struct _xicmp_hdr_t {
    uint8_t type;           // 类型
    uint8_t code;           // 代码
    uint16_t checksum;      // ICMP 报文的校验和
    uint16_t id;            // 标识符
```

```
    uint16_t seq;           // 序号
} xicmp_hdr_t;
```

该结构体定义了 ICMP 数据报的各个字段，包括：

- `type`：ICMP 消息类型，例如回显请求（`8`）和回显响应（`0`）。
- `code`：ICMP 消息代码，用于进一步细分消息类型。
- `checksum`：ICMP 报文的校验和，用于检测报文是否损坏。
- `id` 和 `seq`：标识符和序号，用于匹配请求和响应。

此外，我们还定义了 ICMP 消息类型的常量：

```
#define XICMP_CODE_ECHO_REQUEST             8           // 回显请求
#define XICMP_CODE_ECHO_REPLY               0           // 回显响应
```

4.4.2 实现 ICMP 输入

ICMP 输入函数 `xicmp_in` 负责处理接收到的 ICMP 数据报。在 `xnet_tiny.c` 中，我们实现了该函数：

```
void xicmp_in(xipaddr_t *src_ip, xnet_packet_t * packet) {
  xicmp_hdr_t* icmphdr = (xicmp_hdr_t *)packet->data; // 获取 ICMP 头部指针
  if (
      (packet->size >= sizeof(xicmp_hdr_t))
      && (icmphdr->type == XICMP_CODE_ECHO_REQUEST)
    )
  {
    reply_icmp_request(icmphdr, src_ip, packet); // 如果是 ECHO 请求，发送 ECHO 回复
  }
}
```

该函数的主要功能是：

1. 检查接收到的 ICMP 数据报是否完整。
2. 如果 ICMP 消息类型为回显请求（`XICMP_CODE_ECHO_REQUEST`），则调用 `reply_icmp_request` 函数发送回显响应。

4.4.3 实现 ICMP-ping 响应

ICMP-ping 响应函数 `reply_icmp_request` 负责生成并发送 ICMP 回显响应。在 `xnet_tiny.c` 中，我们实现了该函数：

```
static xnet_err_t reply_icmp_request( xicmp_hdr_t * icmp_hdr,
                                      xipaddr_t* src_ip,
                                      xnet_packet_t * packet)
{
    xicmp_hdr_t * replay_hdr;
    xnet_packet_t * tx = xnet_alloc_for_send(packet->size);

    replay_hdr = (xicmp_hdr_t *)tx->data; // 获取 ICMP 头部指针
    replay_hdr->type = XICMP_CODE_ECHO_REPLY; // 设置 ICMP 类型为 ECHO 回复
```

```
    replay_hdr->code = 0; // 设置代码为 0
    replay_hdr->id = icmp_hdr->id; // 复制 ID
    replay_hdr->seq = icmp_hdr->seq; // 复制序列号
    replay_hdr->checksum = 0; // 初始化校验和字段
    // 复制数据部分
    memcpy(
            ((uint8_t *)replay_hdr)+ sizeof(xicmp_hdr_t),
            ((uint8_t *)icmp_hdr) + sizeof(xicmp_hdr_t),
            packet->size - sizeof(xicmp_hdr_t)
          );
    // 计算校验和
    replay_hdr->checksum = checksum16((uint16_t*)replay_hdr, tx->size, 0, 1);
    return xip_out(XNET_PROTOCOL_ICMP, src_ip, tx); // 发送 ICMP 回复包
}
```

该函数的主要功能是：

1. 分配一个新的数据包用于发送 ICMP 回显响应。
2. 设置 ICMP 回显响应的各个字段，包括类型、代码、ID、序列号和校验和。
3. 复制原始 ICMP 请求的数据部分到响应中。
4. 计算并设置 ICMP 回显响应的校验和。
5. 通过 IP 层发送 ICMP 回显响应。

通过该函数，我们可以实现对 ICMP-ping 请求的响应，从而支持基本的网络连通性测试。

下面进行测试。重新编译后，按照以下流程进行测试：

- 开启 Wireshark 抓包；
- 运行本程序；
- 在虚拟机上 ping 本程序，以触发 ICMP-ping 请求；
- 在虚拟机上查看 ping 结果，看是否 ping 通；
- 查看 Wireshark 抓包结果（见下页）。

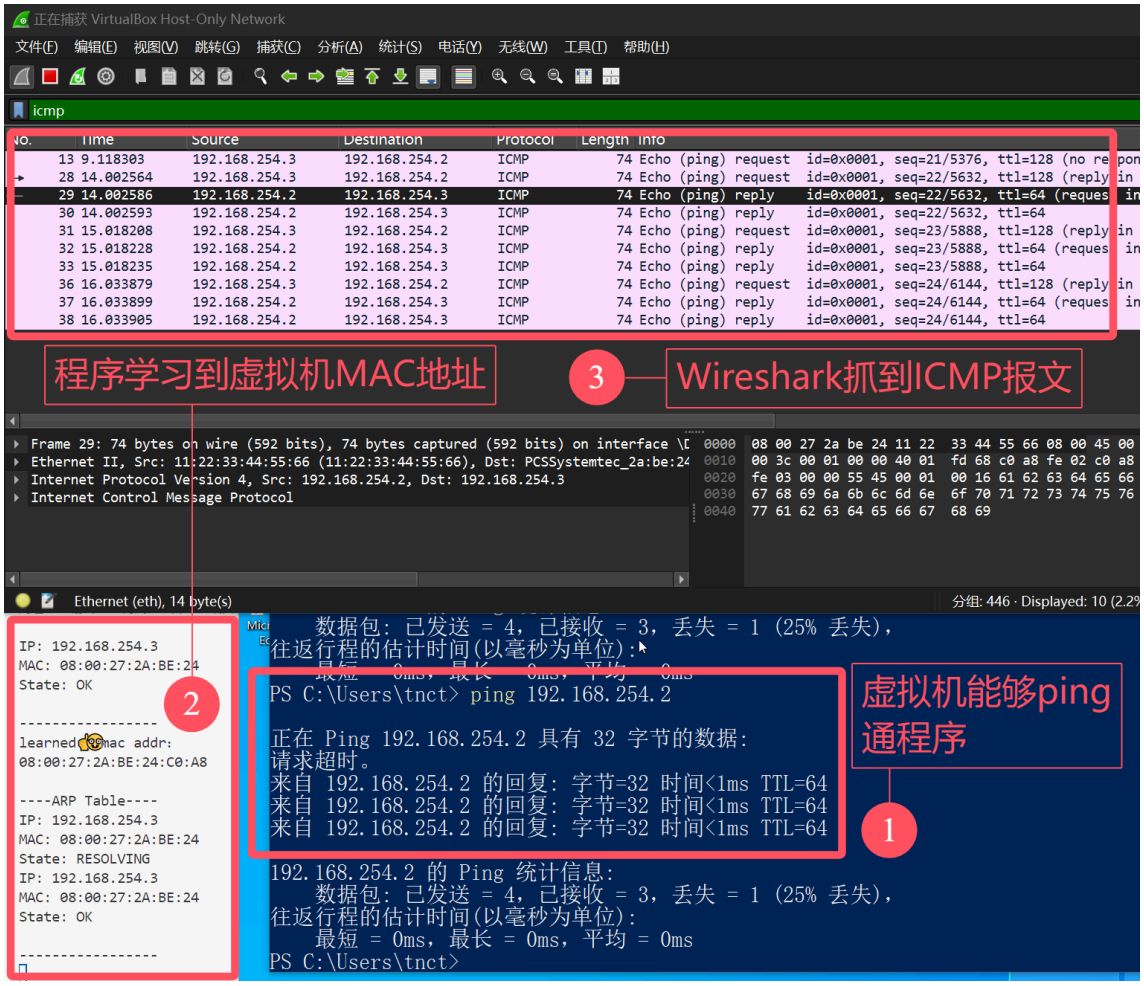

Figure 10: ICMP-ping 响应

从 Figure 10 中可以看到，ICMP-ping 响应发送成功，程序输出中也表明学习到了虚拟机的 MAC 地址，说明代码编写正确。

4.5 关于多 ARP 表项

代码在定义相关数据结构时（见 Section 4.2.1 ），已经考虑到了多 ARP 表项的情况。在函数的具体实现中，均是使用循环遍历整个 ARP 表来进行表项的操作。

在 Figure 8、Figure 10 中的程序输出中，均有 2 个表项（对同一个 ARP 表项的不同状态）。如果考虑再添加一台虚拟机，则可以在输出中看到更多表项。但本次实验所用物理机内存有限，无法同时运行 2 个虚拟机，所以不再演示。

5 实验总结

5.1 内容总结

本次实验主要围绕 ARP 协议的实现展开，通过编写程序完善了 TCP/IP 协议栈的 ARP 协议部分。实验内容包括 ARP 的初始化、无回报 ARP 的生成、ARP 的输入处理以及 ARP 的超时重新请求机制。在基础任务中，成功实现了 ARP 协议的核心功能，包括 ARP 请求与响应的构

建与解析、ARP 缓存表的管理等。此外，还完成了拓展任务，实现了多个 ARP 表项的管理以及 IP 层的输入输出处理。

在实验过程中，首先通过 Wireshark 抓包分析了 ARP 报文的结构，并基于此定义了 ARP 报文的数据结构和相关函数。接着，实现了 ARP 报文的发送与接收功能，包括 ARP 请求的生成与广播、ARP 响应的处理以及 ARP 缓存表的更新。通过定时器机制，实现了 ARP 表项的超时重传功能，确保了 ARP 缓存的准确性和及时性。

在完成 ARP 协议的基础上，进一步实现了 IP 协议和 ICMP 协议。通过定义 IP 数据报和 ICMP 数据报的结构，实现了 IP 层的输入输出功能，并成功实现了 ICMP 的 ping 响应功能。实验结果表明，ARP、IP 和 ICMP 协议的功能均得到了正确实现，能够有效支持网络设备之间的通信。

5.2 心得感悟

通过本次实验，我深刻理解了 ARP 协议的工作原理及其在网络通信中的重要作用。ARP 协议作为 TCP/IP 协议栈中的重要组成部分，负责将 IP 地址解析为 MAC 地址，是网络设备之间通信的基础。通过亲手实现 ARP 协议的核心功能，我不仅加深了对 ARP 协议的理解，还掌握了网络协议栈的实现方法。

在实验过程中，我遇到了一些挑战，例如如何正确解析 ARP 报文、如何管理 ARP 缓存表以及如何处理 ARP 超时重传等。通过查阅资料、分析抓包数据以及反复调试代码，我逐步解决了这些问题，并成功实现了 ARP 协议的功能。这让我认识到，网络协议的实现不仅需要扎实的理论基础，还需要细致的调试和问题解决能力。

此外，通过实现 IP 和 ICMP 协议，我进一步了解了网络层协议的工作原理。IP 协议负责数据包的传输，而 ICMP 协议则用于传递控制消息。通过实现 ICMP 的 ping 响应功能，我掌握了 ICMP 协议的基本实现方法，并理解了其在网络连通性测试中的应用。

总的来说，本次实验让我对 TCP/IP 协议栈有了更深入的理解，并提升了我的编程能力和网络协议分析能力。这些知识和技能对我今后学习更复杂的网络协议以及从事网络相关工作具有重要意义。

参考文献

[1] 未知作者.【一学就会】(一) C++编译工具链——基于 VSCode 的 CMake、make 与 g++简单理解与应用示例[EB/OL](2024-12-31). https://blog.csdn.net/weixin_46248871/article/details/137500744

[2] 李吱恩. cmake 使用详细教程（日常使用这一篇就足够了）[EB/OL](2023-11-28). https://blog.csdn.net/iuu77/article/details/129229361

[3] JON POSTEL. INTERNET PROTOCOL[EB/OL](1981-09). https://www.rfc-editor.org/rfc/rfc791

[4] 刘玉坦. windows 网络命令：ping、ipconfig、tracert、netstat、arp[EB/OL](2024-12-31). https://www.cnblogs.com/liuyutan/p/13289747.html

[5] 哔哩哔哩用户. c03.00 IP 层的输入处理(2)[EB/OL](2024-12-31). https://www.bilibili.com/video/BV1KL4y1P7vG