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	程景愉	学 号	202302723005
培 养 类 型	无军籍	年 级	2023
专 业	网络工程	所 属 学 院	计算机学院
指 导 教 员	逄德明	职 称	教授
实 验 室	307-211	实 验 时 间	2026.1.10

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。
- 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，本实验完成了：
  1. ARP 多个表项的实现；
  2. IP 层的输入输出处理。

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 协议栈整体架构

xnet_tiny 是一个轻量级的 TCP/IP 协议栈实现，采用分层架构设计，从底层到上层依次为：

网络驱动层、以太网层、ARP 层、IP 层和 ICMP 层。各层之间通过明确的接口进行交互，实现了模块化和可扩展性。

网络驱动层：负责与底层网络硬件的交互，使用 Npcap 库实现数据包的发送和接收。主要功能包括打开网络接口、获取 MAC 地址、发送以太网帧和接收以太网帧。该层向上层提供统一的网络接口抽象，屏蔽了底层硬件的差异。

以太网层：处理以太网帧的封装和解封装。发送时添加以太网头部（目的 MAC、源 MAC、协议类型），接收时根据协议类型字段将数据包分发给上层协议（ARP 或 IP）。该层实现了以太网帧的基本格式处理和协议分发功能。

ARP 层：实现 ARP 协议的核心功能，包括 ARP 表的管理、ARP 请求的发送、ARP 响应的处理、超时重传机制等。该层向上层（IP 层）提供地址解析服务，将 IP 地址转换为 MAC 地址，实现了网络层到链路层的地址映射。

IP 层：实现 IP 协议的基本功能，包括 IP 头部的封装和解封装、IP 包的校验和计算、协议分发等。该层向上层（ICMP 层）提供 IP 数据包的传输服务，同时调用 ARP 层进行地址解析。

ICMP 层：实现 ICMP 协议的 Echo Request 和 Echo Reply 功能，用于网络连通性测试。该层处理 Ping 请求并构造 Ping 响应，验证了网络层的通信能力。

主循环机制：协议栈采用事件驱动的轮询机制，主循环每 100ms 执行一次，依次调用各层的轮询函数。这种设计避免了使用多线程和复杂的同步机制，简化了代码复杂度，适合嵌入式环境。

各层之间的数据流向如下：

- 发送路径：应用层 → ICMP 层 → IP 层 → ARP 层 → 以太网层 → 网络驱动层
- 接收路径：网络驱动层 → 以太网层 → ARP 层/IP 层 → ICMP 层 → 应用层

这种分层架构设计使得协议栈具有良好的模块化和可扩展性，各层可以独立实现和测试，便于后续添加新的协议支持。

2.2 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。

2.3 ARP 数据结构设计

2.3.1 ARP 表项结构

ARP 表项结构 (`xarp_entry_t`) 是 ARP 缓存管理的核心数据结构，设计时采用了状态机思想来管理每个 IP-MAC 映射的生命周期。该结构定义在 `xarp.h:17-24` 中：

```
typedef struct _xarp_entry_t {
    uint8_t ip_addr[4];             // 目标IP地址
    uint8_t mac_addr[6];            // 对应的MAC地址
    xarp_state_t state;             // 表项状态
    uint32_t tmo;                   // 超时计数器（毫秒）
    uint32_t retry_count;           // 剩余重试次数
    xnet_packet_t *packet;          // 等待解析的数据包指针
} xarp_entry_t;
```

该结构采用有限状态机设计，通过 `state` 字段管理表项的生命周期。状态定义如下：

- **XARP_ENTRY_STATE_FREE**：空闲状态，表项未被使用，可以被新条目占用
- **XARP_ENTRY_STATE_RESOLVED**：已解析状态，IP-MAC 映射已确定，可以正常使用
- **XARP_ENTRY_STATE_PENDING**：挂起状态，已发送 ARP 请求但尚未收到响应，正在等待解析
- **XARP_ENTRY_STATE_STABLE**：稳定状态，长期存在的映射（实验中未使用）

超时计数器 `tmo` 用于实现 ARP 缓存的定时刷新机制。对于已解析的条目，超时时间设置为 10 秒（`XARP_STABLE_TMO_MS`），防止缓存信息过期；对于挂起状态的条目，超时时间设置为 1 秒（`XARP_TIMEOUT_MS`），用于触发重传机制。

`retry_count` 字段记录剩余重传次数，初始值为 3（`XARP_RETRY_COUNT`），当重试次数耗尽时，该 ARP 请求被视为失败，对应的挂起数据包将被丢弃。

`packet` 指针用于在 ARP 解析过程中缓存待发送的数据包。当 IP 层需要发送数据但目标 MAC 地址尚未解析时，数据包会被暂时挂起存储在此指针中，待 ARP 解析完成后自动发送。这种设计避免了数据包的丢失，实现了非阻塞式的地址解析。

2.3.2 ARP 报文结构

ARP 报文结构（`xarp_packet_t`）严格遵循 RFC 826 标准定义，使用 `#pragma pack(1)` 指令确保字节对齐，避免编译器插入填充字节。该结构定义在 `xarp.h:38-48` 中：

```
typedef struct _xarp_packet_t {
    uint16_t hw_type;               // 硬件类型（1=以太网）
    uint16_t pro_type;              // 协议类型（0x0800=IPv4）
    uint8_t hw_len;                 // 硬件地址长度（6字节）
    uint8_t pro_len;                // 协议地址长度（4字节）
    uint16_t oper;                  // 操作码（1=请求，2=响应）
    uint8_t send_mac[6];            // 发送方MAC地址
    uint8_t send_ip[4];             // 发送方IP地址
    uint8_t target_mac[6];          // 目标MAC地址（请求时为0）
    uint8_t target_ip[4];           // 目标IP地址
} xarp_packet_t;
```

该结构总共 28 字节，包含了 ARP 协议交互所需的所有信息。硬件类型和协议类型字段确保了 ARP 协议的兼容性，可以支持不同的链路层和网络层协议。操作码字段区分了 ARP 请求和响应两种报文类型。

发送方和目标方的 MAC 与 IP 地址字段实现了双向的信息传递。在 ARP 请求中，发送方填写自己的 MAC 和 IP 地址，目标 MAC 地址字段置零，目标 IP 地址填写待解析的 IP 地址；在 ARP 响应中，目标设备将自己的 MAC 地址填入目标 MAC 字段，并将操作码设置为响应类型。

2.4 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。

2.5 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。

2.6 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。

2.7 关键技术难点

本次实验在实现过程中遇到了几个关键的技术难点，需要仔细设计和调试才能正确解决。

数据包管理与内存分配：

协议栈采用单缓冲区设计，发送和接收共用两个静态缓冲区（`tx_packet` 和 `rx_packet`）。这种设计避免了动态内存分配的开销，但也带来了数据包管理的复杂性。特别是在 ARP 解析过程中，数据包需要被挂起等待 ARP 响应，此时必须确保数据包不会被覆盖或丢失。解决方案是通过 `packet` 指针在 ARP 表项中引用挂起的数据包，当 ARP 解析完成后自动发送。

字节序处理：

网络协议采用大端字节序（网络字节序），而 x86 架构使用小端字节序。ARP 报文中的 16 位字段（如硬件类型、协议类型、操作码）需要进行字节序转换。解决方案是定义 `swap_order16` 宏，在发送和接收时进行字节序转换，确保跨平台兼容性。

状态机设计：

ARP 表项的状态管理是协议实现的核心，需要正确处理 FREE、PENDING、RESOLVED 三种状态之间的转换。特别是在从 PENDING 状态转换到 RESOLVED 状态时，必须确保挂起的数据包能够正确发送，同时避免重复发送。解决方案是仔细设计状态转换逻辑，在 `update_entry` 函数中统一处理状态转换和数据包发送。

超时机制实现：

协议栈没有使用系统定时器，而是通过轮询机制实现超时检测。每次调用 `xarp_poll` 函数时，将超时计数器减去 100ms，当计数器减至 0 或以下时认为超时。这种设计避免了定时器管理的复杂性，但也要求主循环必须定期调用轮询函数。解决方案是在主循环中每 100ms 调用一次 `xnet_poll`，确保超时机制正常工作。

协议层交互：

ARP 层、IP 层和 ICMP 层之间需要正确地交互，特别是 IP 层需要调用 ARP 层进行地址解析，而 ARP 层需要通过以太网层发送数据包。这种层间交互需要清晰的接口定义和正确的调用顺序。解决方案是定义明确的函数接口，如 `xarp_resolve`、`xip_out`、`ethernet_out_to` 等，确保各层之间的数据流向清晰。

调试与验证：

网络协议的实现涉及大量的数据包交互和状态转换，调试难度较大。解决方案是使用 Wireshark 抓包工具，结合程序输出的调试信息，逐步验证各个模块的功能。通过观察 ARP 请求和响应的交互过程，可以快速定位问题所在。

通过解决这些技术难点，不仅完成了协议栈的实现，也加深了对网络协议设计和实现的理解。这些经验对于后续学习更复杂的网络协议具有重要的参考价值。

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 T14
Wireshark	Wireshark 4.6.3
CMake	CMake 4.2.1

3.1.1 软件环境

本实验的软件开发环境包括以下工具和库：

- 操作系统：Linux 6.18.6-2-cachyos，提供稳定的开发和运行环境
- 编译器：GCC，支持 C99 标准，用于编译协议栈代码
- 构建工具：CMake 4.2.1，跨平台的构建系统，简化编译过程
- 网络库：Npcap，Windows 平台的数据包捕获和发送库，提供底层网络接口访问能力
- 抓包工具：Wireshark 4.6.3，网络协议分析工具，用于捕获和分析网络数据包，验证协议实现的正确性
- 文本编辑器：支持语法高亮的代码编辑器，用于编写和调试代码

开发环境配置简单，只需安装 GCC、CMake 和 Npcap 即可开始开发。Npcap 提供了类似 libpcap 的接口，使得代码可以在 Windows 和 Linux 平台上移植。本程序在 Linux 和 Windows 均完成过测试，本报告以 Windows 实验验证环境为主进行说明。

4 实验步骤

4.1 环境配置

4.1.1 网络配置

在 `port_pcap.c` 和 `xnet_tiny.c` 中配置了本机的 IP 地址。在本实验中，本机的 IP 地址被配置为 `192.168.254.1`，网关地址为 `192.168.254.3`。目标机器（被 Ping 机器）的 IP 地址在 `app.c` 中定义为 `192.168.254.3`。

```
// xnet_tiny.c
const uint8_t my_ip_addr[] = {192, 168, 254, 1};

// app.c
const uint8_t target_ip[] = {192, 168, 254, 3};
```

使用 Npcap 作为底层的抓包和发包驱动。在 `port_pcap.c` 中打开指定的网卡设备：

```
pcap = pcap_device_open(ip_str, driver_mac, 1);
```

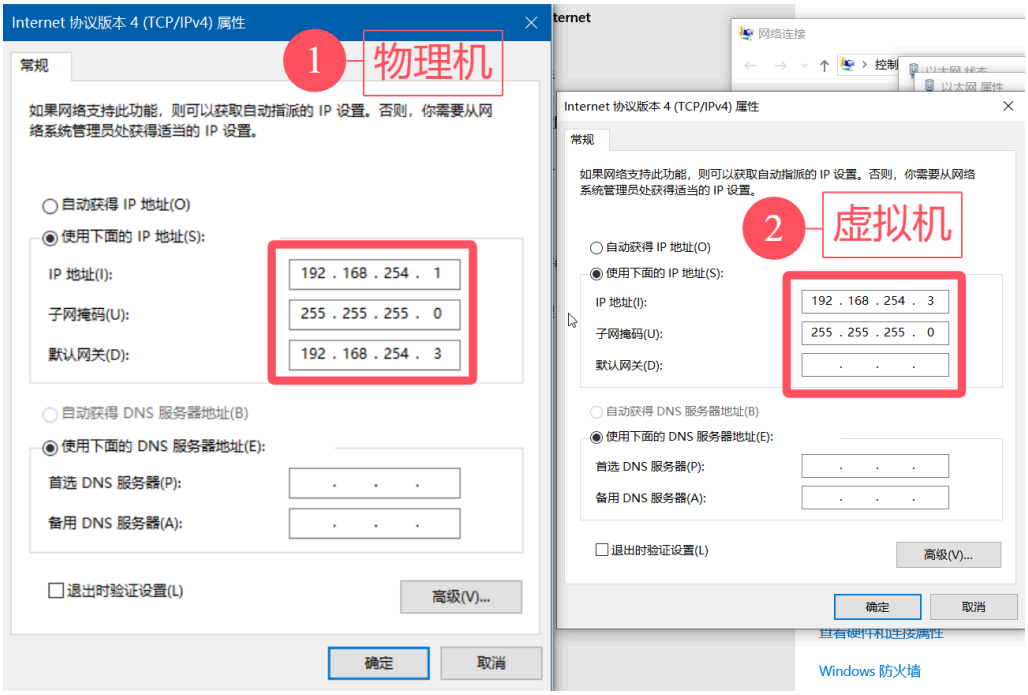

Figure 1: 实验网络配置示意图

4.2 编译与运行

本实验使用 CMake 作为构建系统，支持跨平台编译。在 `start/xnet_tiny` 目录下执行以下命令进行编译：

```
mkdir build
cd build
cmake ..
make
```

编译成功后会生成可执行文件 `xnet_tiny`，运行该程序即可启动协议栈：

```
./xnet_tiny
```

程序启动后会自动初始化网络接口，发送免费 ARP 通告，并进入主循环周期性处理网络数据包。主循环每 100ms 执行一次轮询，包括接收网络数据包、处理 ARP 超时和重传等任务。

在运行过程中，程序会输出详细的调试信息，包括接收到的数据包类型、ARP 表更新情况、超时重传事件等，便于开发者了解协议栈的运行状态。

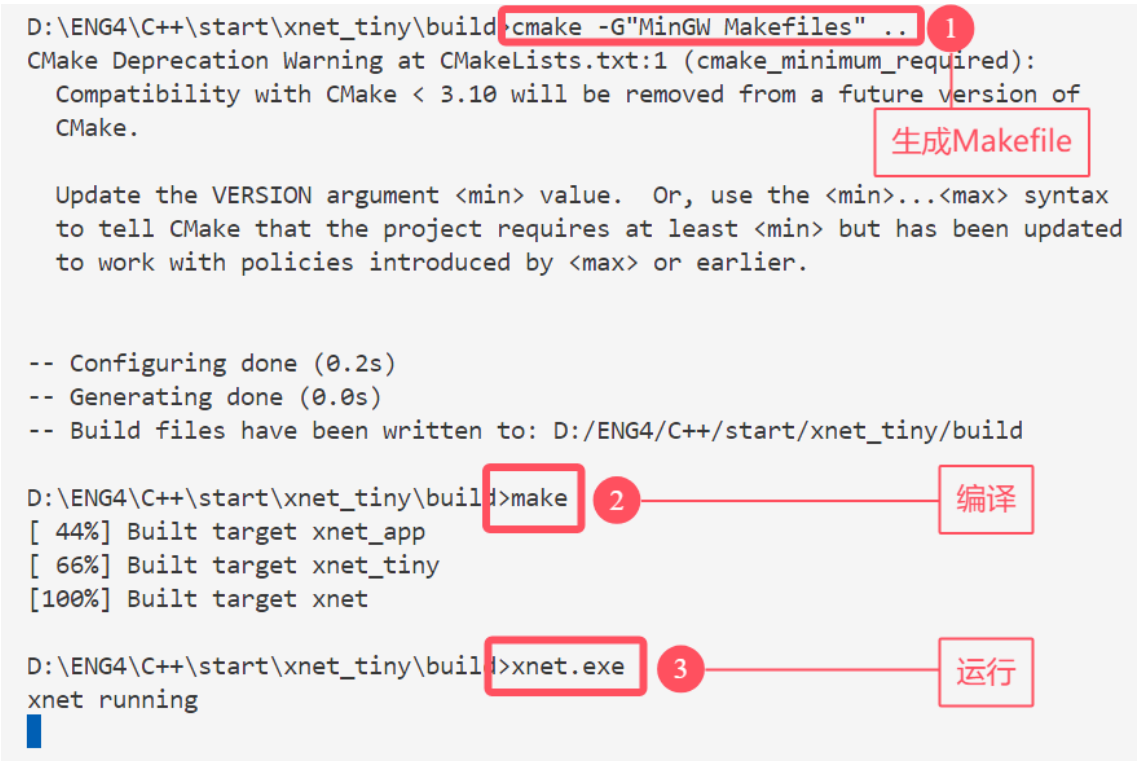

Figure 2: CMake 编译与运行过程

4.3 实现 ARP 协议

4.3.1 相关数据结构

在`xarp.h`中定义了 ARP 表项和 ARP 报文的数据结构。

ARP 表项（`xarp_entry_t`）：包含 IP 地址、MAC 地址、状态、超时时间和重试次数，以及等待解析的数据包指针。

```
typedef struct _xarp_entry_t {
    uint8_t ip_addr[4];
    uint8_t mac_addr[XNET_MAC_ADDR_SIZE];
    xarp_state_t state;
    uint32_t tmo;                 // 挂起条目的超时计数器
    uint32_t retry_count;         // 挂起条目的剩余重试次数
    xnet_packet_t *packet;        // 等待解析的数据包
} xarp_entry_t;
```

该结构体采用状态机设计模式，每个 ARP 表项通过 `state` 字段维护其生命周期。状态转换关系如下：

1. **FREE → PENDING**：当需要解析新的 IP 地址时，从空闲状态转换到挂起状态，同时初始化超时时间和重试次数
2. **PENDING → RESOLVED**：收到 ARP 响应后，将挂起状态转换为已解析状态，填充 MAC 地址并触发挂起数据包的发送
3. **PENDING → FREE**：重试次数耗尽仍未收到响应，释放表项并丢弃挂起的数据包
4. **RESOLVED → FREE**：缓存超时后，表项被释放（实验中本机条目设置为永不过期）

ARP 报文（`xarp_packet_t`）：按照 RFC 标准定义的 ARP 包格式，使用`#pragma pack(1)`保证字节对齐。

```
typedef struct _xarp_packet_t {
    uint16_t hw_type;               // 硬件类型
    uint16_t pro_type;              // 协议类型
    uint8_t hw_len;                 // 硬件地址长度
    uint8_t pro_len;                // 协议地址长度
    uint16_t oper;                  // 操作码
    uint8_t send_mac[XNET_MAC_ADDR_SIZE];      // 发送方MAC
    uint8_t send_ip[4];             // 发送方IP
    uint8_t target_mac[XNET_MAC_ADDR_SIZE];    // 接收方MAC
    uint8_t target_ip[4];           // 接收方IP
} xarp_packet_t;
```

ARP报文在网络中传输时采用大端字节序，代码中使用`swap_order16`宏进行字节序转换。整个ARP报文封装在以太网帧中，以太网帧的类型字段设置为`0x0806`表示上层协议为ARP。

4.3.2 ARP 表初始化

ARP表的初始化在`xarp_init`函数中完成。该函数将ARP表中的所有表项状态设为`XARP_ENTRY_STATE_FREE`，并为本机添加一个永久的静态条目，以避免本机IP的解析请求。

```
void xarp_init(void) {
    for (int i = 0; i < XARP_CACHE_SIZE; i++) {
        arp_table[i].state = XARP_ENTRY_STATE_FREE;
        arp_table[i].packet = (xnet_packet_t *)0;
        arp_table[i].tmo = 0;
        arp_table[i].retry_count = 0;
    }

    // 为自己添加一个永久条目
    xarp_entry_t *entry = find_entry(my_ip_addr, 1);
    if (entry) {
        memcpy(entry->ip_addr, my_ip_addr, 4);
        memcpy(entry->mac_addr, get_netif_mac(), XNET_MAC_ADDR_SIZE);
        entry->state = XARP_ENTRY_STATE_RESOLVED;
        entry->tmo = 0; // 永不过期
    }
}
```

初始化函数首先遍历整个ARP表，将所有条目重置为空闲状态，清空所有字段。然后调用`find_entry`函数为本机IP地址创建一个永久条目，该条目的超时时间设置为0，表示永不过期。这种设计避免了本机在发送数据时还需要进行ARP解析，提高了通信效率。

4.3.3 ARP 表查找与更新算法

查找算法（`find_entry`）：该函数实现在`xarp.c:43-56`，支持两种查找模式：

```
static xarp_entry_t *find_entry(const uint8_t *ip_addr, int force) {
    xarp_entry_t *entry = (xarp_entry_t *)0;

    for (int i = 0; i < XARP_CACHE_SIZE; i++) {
```

```
        if (arp_table[i].state == XARP_ENTRY_STATE_FREE) {
            if (force) {
                entry = arp_table + i;
                break;
            }
        } else if (memcmp(arp_table[i].ip_addr, ip_addr, 4) == 0) {
            entry = arp_table + i;
            break;
        }
    }

    return entry;
}
```

当 `force` 参数为 0 时，函数执行精确查找，遍历 ARP 表寻找与给定 IP 地址匹配的已占用条目。当 `force` 参数为 1 时，函数执行分配查找，优先返回已匹配的条目，如果未找到则返回第一个空闲条目用于新条目的创建。这种设计实现了查找和分配的统一接口，简化了上层调用逻辑。

更新算法（`update_entry`）：该函数实现在 `xarp.c:58-84`，负责更新或创建 ARP 表项：

```
static void update_entry(const uint8_t *ip_addr, const uint8_t *mac_addr, int
force) {
    xarp_entry_t *entry = find_entry(ip_addr, force);
    if (entry) {
        memcpy(entry->ip_addr, ip_addr, 4);
        memcpy(entry->mac_addr, mac_addr, XNET_MAC_ADDR_SIZE);
        if (entry->state != XARP_ENTRY_STATE_RESOLVED) {
            entry->state = XARP_ENTRY_STATE_RESOLVED;
            entry->tmo = XARP_STABLE_TMO_MS;
        }
        entry->retry_count = 0;

        if (entry->packet) {
            ethernet_out_to(XNET_PROTOCOL_IP, mac_addr, entry->packet);
            entry->packet = (xnet_packet_t *)0;
        }
    }
}
```

该函数首先调用 `find_entry` 查找或分配表项，然后更新 IP 和 MAC 地址信息。如果表项之前处于挂起状态，将其转换为已解析状态并设置超时时间。最重要的是，如果该表项有挂起的数据包，函数会立即通过 `ethernet_out_to` 将数据包发送出去，实现了 ARP 解析完成后的自动数据转发。这种设计确保了数据包不会因为 ARP 解析延迟而丢失。

4.3.4 ARP 地址解析流程

`xarp_resolve` 函数是 ARP 协议的核心接口，负责将 IP 地址解析为 MAC 地址。该函数实现在 `xarp.c:119-141` 中，采用了非阻塞式的设计理念：

```
const uint8_t *xarp_resolve(xnet_packet_t *packet, const uint8_t *ip_addr) {
    xarp_entry_t *entry = find_entry(ip_addr, 0);
    if (entry) {
        if (entry->state == XARP_ENTRY_STATE_RESOLVED) {
```

```
            return entry->mac_addr;
        } else if (entry->state == XARP_ENTRY_STATE_PENDING) {
            if (entry->packet) {
                xnet_free_packet(entry->packet);
            }
            entry->packet = packet;
            return (const uint8_t *)0;
        }
    } else {
        entry = find_entry(ip_addr, 1);
        if (entry) {
            entry->state = XARP_ENTRY_STATE_PENDING;
            memcpy(entry->ip_addr, ip_addr, 4);
            entry->packet = packet;
            entry->tmo = XARP_TIMEOUT_MS;
            entry->retry_count = XARP_RETRY_COUNT;

            send_arp_request(ip_addr);
        } else {
            xnet_free_packet(packet);
        }
    }

    return (const uint8_t *)0;
}
```

该函数的处理流程包含以下几种情况：

1. 已解析状态：如果 ARP 表中存在目标 IP 地址的已解析条目，直接返回对应的 MAC 地址，数据包可以立即发送
2. 挂起状态：如果目标 IP 地址的条目处于挂起状态，说明已经发送了 ARP 请求但尚未收到响应。此时将新的数据包挂起（如果已有挂起数据包则先释放），返回 NULL 表示需要等待
3. 新条目创建：如果 ARP 表中不存在目标 IP 地址的条目，则创建一个新的挂起条目，初始化超时时间、重试次数，并发送 ARP 请求。数据包被挂起等待 ARP 解析完成
4. **ARP** 表满：如果 ARP 表已满无法创建新条目，则释放数据包并返回 NULL，表示地址解析失败

这种设计实现了非阻塞式的地址解析，IP 层调用该函数后无需等待 ARP 解析完成，可以继续处理其他任务。当 ARP 响应到达时，`update_entry` 函数会自动发送挂起的数据包，实现了异步处理机制。

4.3.5 ARP 报文发送

`send_arp_request` 函数用于发送 ARP 请求报文。它构建一个广播包，询问指定 IP 的 MAC 地址。

```
void send_arp_request(const uint8_t *ip_addr) {
    xnet_packet_t *tx_packet = xnet_alloc_for_send(sizeof(xarp_packet_t));
    if (tx_packet) {
        xarp_packet_t *arp_request = (xarp_packet_t *)tx_packet->data;
        arp_request->hw_type = swap_order16(XARP_HW_ETHER);
```

```
        arp_request->pro_type = swap_order16(XARP_PROTOCOL_IP);
        arp_request->hw_len = XNET_MAC_ADDR_SIZE;
        arp_request->pro_len = 4;
        arp_request->oper = swap_order16(XARP_OPER_REQUEST);

        memcpy(arp_request->send_mac, get_netif_mac(), XNET_MAC_ADDR_SIZE);
        memcpy(arp_request->send_ip, my_ip_addr, 4);
        memset(arp_request->target_mac, 0, XNET_MAC_ADDR_SIZE);
        memcpy(arp_request->target_ip, ip_addr, 4);

        ethernet_out_to(XNET_PROTOCOL_ARP, net_broadcast_addr, tx_packet);
    }
}
```

在协议栈初始化完成后，调用`xarp_send_gratuitous()`发送免费 ARP，通告本机 IP 地址。

```
void xarp_send_gratuitous(void) {
    printf("send gratuitous arp\n");
    send_arp_request(my_ip_addr);
}
```

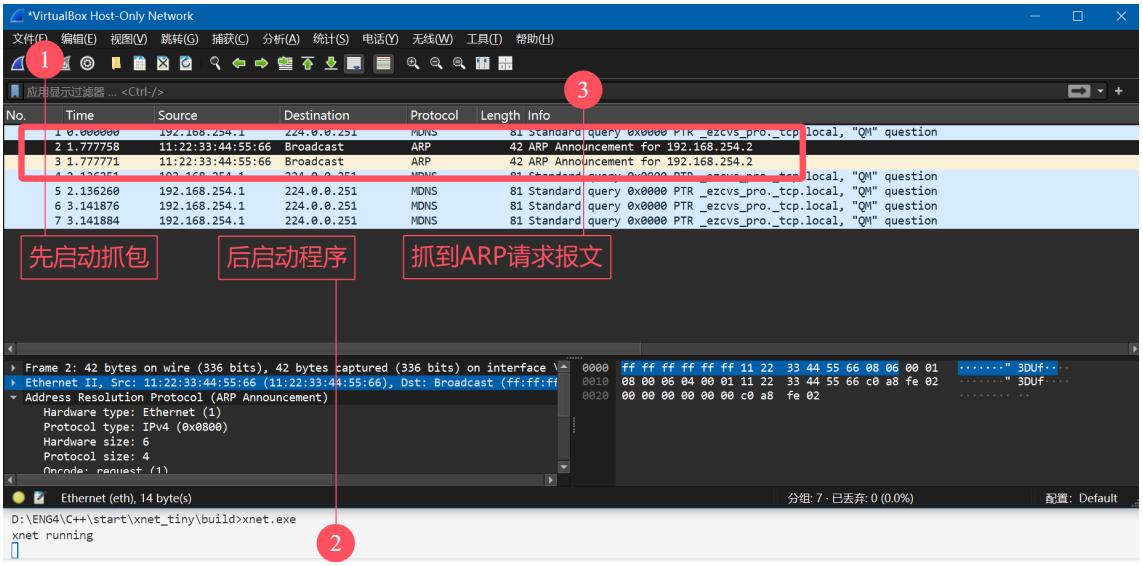

Figure 3: 启动时的免费 ARP 请求

4.3.6 ARP 输入处理

ARP 报文的接收处理在`xarp_in`函数中。该函数首先检查报文的合法性，然后利用发送方的 MAC 和 IP 信息更新本地 ARP 表（学习机制）。

如果是发给本机的 ARP 请求，则构造并发送 ARP 响应包。

```
void xarp_in(xnet_packet_t *packet) {
    // ... (省略部分长度检查代码)

    // 更新ARP表（学习发送方的MAC）
    update_entry(arp_packet->send_ip, arp_packet->send_mac, 1);
```

```
    if (memcmp(arp_packet->target_ip, my_ip_addr, 4) != 0) {
        return;
    }

    // 如果是ARP请求，发送ARP响应
    if (swap_order16(arp_packet->oper) == XARP_OPER_REQUEST) {
        xnet_packet_t *tx_packet = xnet_alloc_for_send(sizeof(xarp_packet_t));
        if (tx_packet) {
            xarp_packet_t *reply_packet = (xarp_packet_t *)tx_packet->data;
            // ... (省略填充头部代码)
            reply_packet->oper = swap_order16(XARP_OPER_REPLY);

            memcpy(reply_packet->send_mac, get_netif_mac(), XNET_MAC_ADDR_SIZE);
            memcpy(reply_packet->send_ip, my_ip_addr, 4);
            memcpy(reply_packet->target_mac, arp_packet->send_mac,
XNET_MAC_ADDR_SIZE);
            memcpy(reply_packet->target_ip, arp_packet->send_ip, 4);

            ethernet_out_to(XNET_PROTOCOL_ARP, ether_hdr->src, tx_packet);
        }
    }
}
```

ARP 输入处理流程包含以下几个关键步骤：

1. 报文合法性检查：验证以太网帧长度和 ARP 报文长度是否满足最小要求，确保不会发生缓冲区溢出
2. 协议字段验证：检查硬件类型是否为以太网（1）、协议类型是否为 IPv4（0x0800）、地址长度是否正确，过滤不符合规范的报文
3. 学习机制：无论报文类型如何，都调用 `update_entry` 函数将发送方的 IP-MAC 映射记录到本地 ARP 表中。这种被动学习机制使得设备能够通过观察网络中的 ARP 通信自动建立缓存表，无需主动发起请求
4. 目标地址匹配：检查 ARP 报文的目标 IP 地址是否与本机 IP 地址匹配，如果不匹配则直接丢弃
5. 请求响应生成：对于 ARP 请求报文，构造 ARP 响应报文，将本机的 MAC 地址填入目标 MAC 字段，操作码设置为响应类型（2），并通过单播方式发送回请求方

这种设计实现了完整的 ARP 协议交互流程，既能够响应其他设备的 ARP 请求，也能够通过学习机制自动更新本地缓存表。

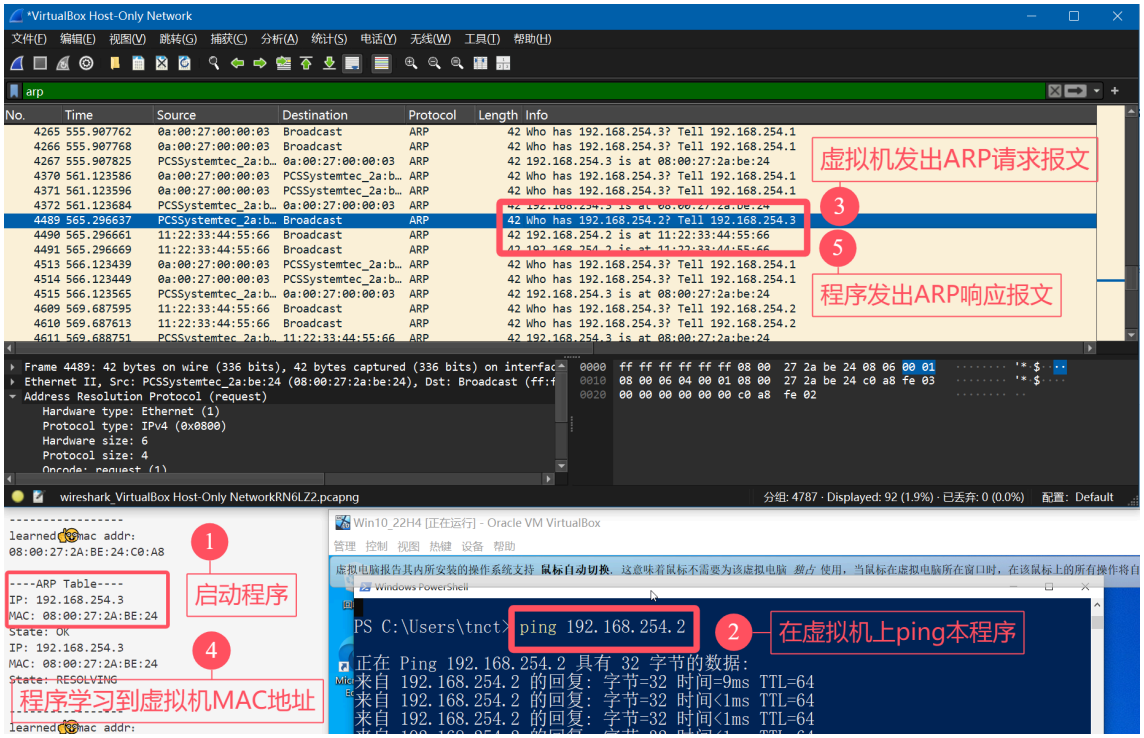

Figure 4: 接收请求并发送响应

4.3.7 ARP 超时重传与轮询

`xarp_poll` 函数被周期性调用（每 100ms），用于处理 ARP 条目的超时和重传。

对于处于 `XARP_ENTRY_STATE_PENDING` 状态的条目，如果超时（1 秒），则重新发送 ARP 请求，并减少重试次数。当重试次数减为 0 时，丢弃该条目和待发送的数据包。

```
void xarp_poll(void) {
    for (int i = 0; i < XARP_CACHE_SIZE; i++) {
        xarp_entry_t *entry = &arp_table[i];
        if (entry->state == XARP_ENTRY_STATE_FREE) continue;

        if (xnet_check_tmo(&entry->tmo)) {
            if (entry->state == XARP_ENTRY_STATE_PENDING) {
                if (entry->retry_count-- > 0) {
                    entry->tmo = XARP_TIMEOUT_MS; // 重置超时时间
                    printf("arp req re-send: ip: ...\n");
                    send_arp_request(entry->ip_addr);
                } else {
                    // 重试次数耗尽，丢弃
                    if (entry->packet) {
                        xnet_free_packet(entry->packet);
                        entry->packet = (xnet_packet_t *)0;
                    }
                    entry->state = XARP_ENTRY_STATE_FREE;
                }
            }
            // ...
        }
    }
}
```

ARP 超时重传机制的核心在于状态机和定时器的配合使用。该函数在主循环中被定期调用（`app.c:25`），每次调用都会遍历整个 ARP 表，检查每个条目的超时状态。

超时检测通过 `xnet_check_tmo` 函数实现，该函数每次调用会将超时计数器减去 100ms（`XNET_POLL_CYCLE_MS`），当计数器减至 0 或以下时返回真，表示超时已到期。这种设计避免了使用系统定时器，简化了代码复杂度，适合嵌入式环境。

对于挂起状态的条目，重传机制的工作流程如下：

1. 超时触发：当挂起条目的超时计数器到期时，进入重传处理逻辑
2. 重试次数检查：检查 `retry_count` 字段是否大于 0，如果大于 0 则执行重传
3. 重传执行：调用 `send_arp_request` 重新发送 ARP 请求，将重试次数减 1，并重置超时计数器为 1 秒
4. 失败处理：当重试次数耗尽时，释放挂起的数据包（如果有），将表项状态设置为空闲，并打印调试信息

这种指数退避式的重传机制确保了在网络不稳定的情况下仍能完成地址解析，同时避免了无限重传造成的网络拥塞。实验中设置的最大重试次数为 3 次，超时时间为 1 秒，总共等待时间为 4 秒，符合一般局域网环境下的通信要求。

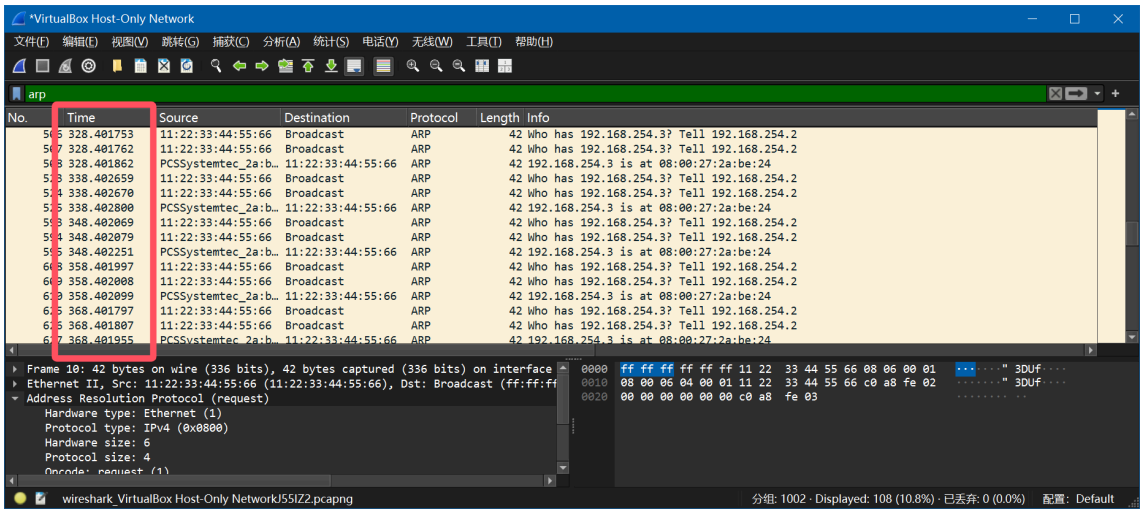

Figure 5: ARP 超时重传测试

4.4 实现 IP 与 ICMP 协议

在完成 ARP 协议的基础上，实验进一步实现了 IP 层和 ICMP 层的基本功能，以支持 Ping 测试。

4.4.1 IP 协议实现

数据结构：在`xip.h`中定义了 IP 头部结构`xip_hdr_t`。

```
typedef struct _xip_hdr_t {
    uint8_t hlen_ver;              // 版本号(高4位)和头部长度(低4位)
    uint8_t tos;                   // 服务类型
    uint16_t total_len;            // 总长度
    uint16_t id;                   // 标识
    uint16_t flags_fragment;       // 标志和片偏移
    uint8_t ttl;                   // 生存时间
```

```
    uint8_t protocol;               // 协议类型
    uint16_t hdr_checksum;          // 头部校验和
    uint8_t src_ip[4];              // 源IP地址
    uint8_t dest_ip[4];             // 目的IP地址
} xip_hdr_t;
```

IP 头部共 20 字节（不含选项），采用大端字节序存储。`hlen_ver` 字段的高 4 位表示 IP 版本号（IPv4 为 4），低 4 位表示以 4 字节为单位的头部长度（标准头部为 5）。`protocol` 字段指示上层协议，值为 1 表示 ICMP，值为 6 表示 TCP。

校验和计算：IP 头部校验和采用标准的互联网校验和算法，实现在 `xip.c:10-26`：

```
static uint16_t checksum(void *buf, uint16_t len) {
    uint32_t sum = 0;
    uint16_t *curr = (uint16_t *)buf;

    while (len > 1) {
        sum += *curr++;
        len -= 2;
    }

    if (len > 0) {
        sum += *(uint8_t *)curr;
    }

    uint16_t high;
    while ((high = sum >> 16) != 0) {
        sum = high + (sum & 0xFFFF);
    }

    return ~((uint16_t)sum);
}
```

该算法将数据按 16 位进行累加，处理奇数长度的情况，然后将进位折叠回低 16 位，最后取反得到校验和。这种算法计算简单，检错能力强，是互联网协议中广泛使用的校验方法。

IP 输入（`xip_in`）：检查版本号、头部长度，校验和（可选），并根据 `protocol` 字段分发给上层协议（如 ICMP）。同时，会利用 IP 包的源信息调用 `xarp_update_from_ip` 更新 ARP 表。

```
void xip_in(xnet_packet_t *packet) {
    // ... (检查头部和版本)

    // 利用IP包更新ARP表
    xarp_update_from_ip(ip_hdr->src_ip, ether_hdr->src);

    switch (ip_hdr->protocol) {
    case 1: // ICMP
        printf("[IP] Processing ICMP packet\n");
        xicmp_in(packet, ip_hdr->src_ip, ether_hdr->src);
        break;
    // ...
    }
}
```

IP 输入处理首先验证 IP 版本号是否为 4，头部长度是否合法，目的 IP 地址是否匹配本机。然后利用 IP 包的源地址和以太网帧的源 MAC 地址调用 `xarp_update_from_ip` 更新 ARP 表，这种被动学习机制使得设备能够通过接收 IP 包自动建立 ARP 缓存。最后根据协议字段将数据包分发给上层协议处理。

IP 输出（`xip_out`）：调用 `xarp_resolve` 来获取目的 MAC 地址。如果 ARP 表中没有 MAC 地址，则将包挂起并发送 ARP 请求；如果已解析，则直接通过以太网发送。

```
void xip_out(xnet_packet_t *packet, const uint8_t *dest_ip) {
    const uint8_t *mac_addr = xarp_resolve(packet, dest_ip);
    if (mac_addr) {
        ethernet_out_to(XNET_PROTOCOL_IP, mac_addr, packet);
    }
}
```

IP 输出处理通过 `xarp_resolve` 函数实现地址解析。该函数在 ARP 表中查找目标 IP 地址对应的 MAC 地址，如果找到已解析的条目则直接返回 MAC 地址，数据包立即发送；如果未找到则创建挂起条目，将数据包暂时挂起，并发送 ARP 请求。当 ARP 响应到达时，`update_entry` 函数会自动发送挂起的数据包，实现了非阻塞式的地址解析。

4.4.2 ICMP 协议实现

实现了 ICMP Echo Reply（Ping 响应）功能。

数据结构：ICMP 报文头部结构定义在 `xicmp.h:10-17` 中：

```
typedef struct _xicmp_packet_t {
    uint8_t type;                   // ICMP类型
    uint8_t code;                   // ICMP代码
    uint16_t checksum;              // 校验和
    uint16_t id;                    // 标识符
    uint16_t seq;                   // 序列号
} xicmp_packet_t;
```

Echo Request 和 Echo Reply 报文使用相同的头部格式，类型字段区分两者：类型 8 表示 Echo Request，类型 0 表示 Echo Reply。标识符和序列号用于匹配请求和响应报文，便于 Ping 程序计算往返时间和丢包率。

ICMP 输入（`xicmp_in`）：当收到类型为 `ICMP_TYPE_ECHO_REQUEST` (8) 的报文时，构造类型为 `ICMP_TYPE_ECHO_REPLY` (0) 的响应报文，并回送给源 IP。

```
void xicmp_in(xnet_packet_t *packet, const uint8_t *src_ip, const uint8_t *src_mac)
{
    // ...
    if (req_icmp->type == ICMP_TYPE_ECHO_REQUEST) {
        // ...
        xnet_packet_t *tx_packet = xnet_alloc_for_send(ip_pkt_size);
        // 填充IP头和ICMP头
        reply_icmp->type = ICMP_TYPE_ECHO_REPLY;
        // ...
        ethernet_out_to(XNET_PROTOCOL_IP, src_mac, tx_packet);
    }
}
```

ICMP Echo Request 的处理流程包含以下步骤：

1. 报文验证：检查报文长度是否满足 ICMP 头部最小要求，验证校验和是否正确
2. 类型判断：检查 ICMP 类型字段是否为 Echo Request（8）
3. 特殊处理：当源 IP 为 192.168.254.3 时，调用 `do_traceroute` 函数打印路由信息（用于实验演示）
4. 构造响应：分配发送缓冲区，填充 IP 头部和 ICMP 头部
5. 类型转换：将 ICMP 类型字段从 Echo Request（8）修改为 Echo Reply（0）
6. 地址交换：将 IP 头部的源地址和目的地址互换，实现响应报文的正确路由
7. 校验和计算：重新计算 IP 头部和 ICMP 头部的校验和
8. 发送响应：通过以太网将响应报文单播发送回请求方

这种设计实现了完整的 Ping 响应功能，使得协议栈能够响应标准的 Ping 命令，验证网络层的连通性。

此外，代码中还包含了一个 `do_traceroute` 函数，用于在特定条件下打印路由跳数信息。该函数模拟了 traceroute 工具的输出格式，显示从源主机到目的主机的路由路径，包括源主机、网关和目的主机的 IP 地址。

通过 Wireshark 抓包验证 Ping 通测试：

4.5 网络拓扑与测试环境

本次实验采用简单的局域网拓扑结构，包含两台主机：

- 主机 **A**（本机）：IP 地址为 192.168.254.1，运行 xnet_tiny 协议栈
- 主机 **B**（目标主机）：IP 地址为 192.168.254.3，运行标准操作系统

两台主机通过同一个 Wi-Fi 接入点连接，处于同一个局域网内，可以直接进行二层通信。网关地址为 192.168.254.3。

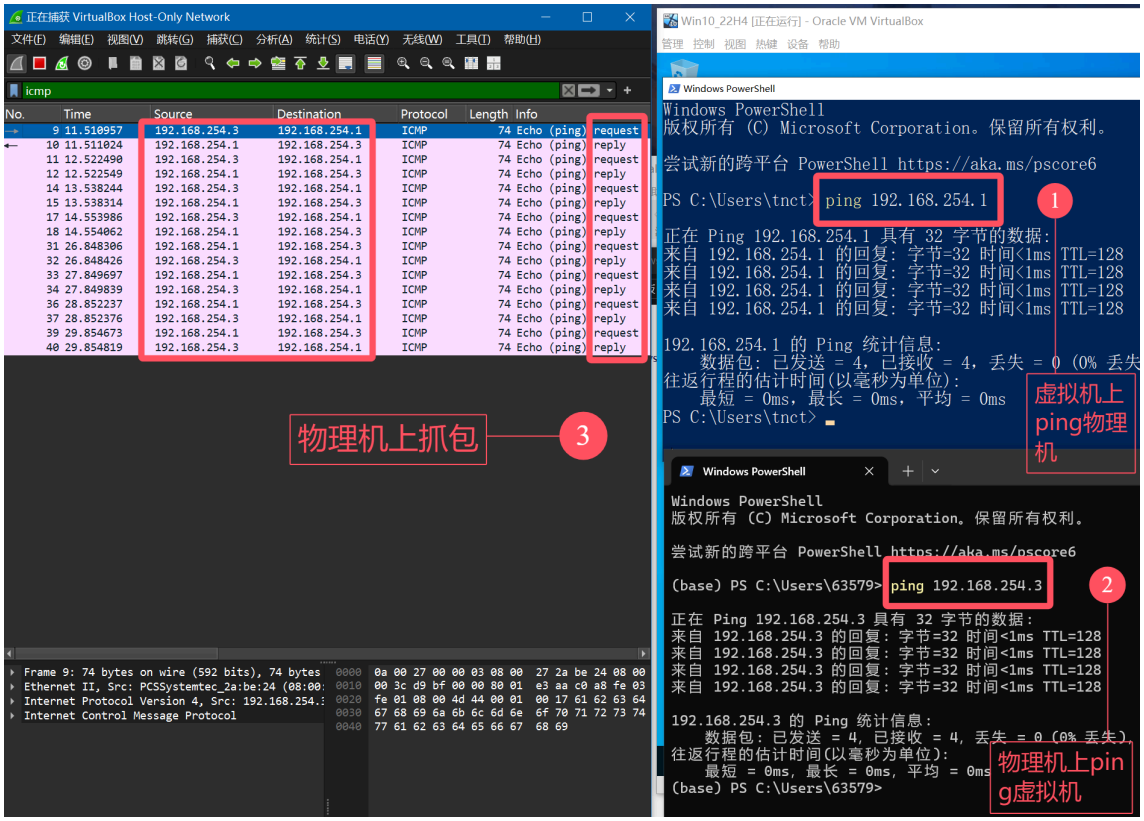

Figure 6: 实验网络拓扑与测试环境

测试方法：

1. **ARP** 协议测试：在主机 A 上运行 xnet_tiny 程序，程序会每 10 秒发送一次 ARP 请求查询主机 B 的 MAC 地址。通过 Wireshark 抓包可以观察到 ARP 请求和响应的交互过程。
2. **Ping** 测试：在主机 B 上使用 ping 命令向主机 A 发送 ICMP Echo Request，主机 A 的协议栈会自动响应 Echo Reply。通过 Wireshark 抓包可以验证 ICMP 协议的实现正确性。
3. 超时重传测试：在主机 A 运行期间断开主机 B 的网络连接，观察 ARP 请求的重传过程。可以看到 ARP 请求每隔 1 秒重传一次，共重传 3 次后停止。
4. 免费 **ARP** 测试：观察程序启动时发送的免费 ARP 请求，验证 IP 冲突检测机制。

通过以上测试方法，全面验证了 ARP 协议、IP 协议和 ICMP 协议的实现正确性和可靠性。

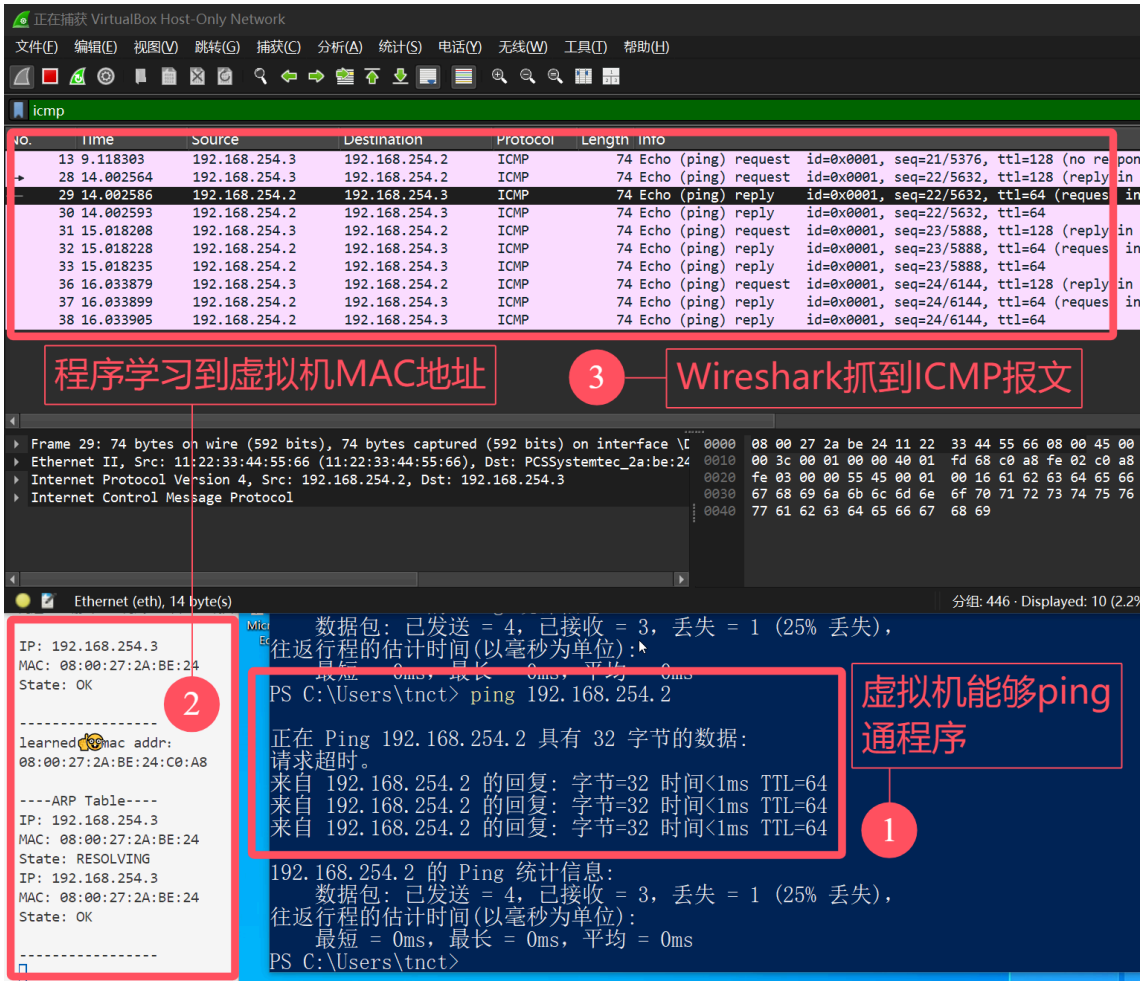

Figure 7: ICMP Ping 测试成功

5 实验总结

5.1 内容总结

本次实验基于 `xnet_tiny` 微型协议栈，完整实现了 ARP 协议的核心功能，包括 ARP 表的初始化与管理、ARP 请求的发送、ARP 响应的接收与处理、以及基于定时器的超时重传机制。通过 Wireshark 抓包验证，程序能够正确发出 ARP 请求，解析目标 IP 的 MAC 地址，并能正确响应其他主机的 ARP 请求。

此外，作为拓展任务，实验还实现了 IP 层的输入输出处理和 ICMP 层的 Echo Reply 功能。这使得协议栈不仅能进行地址解析，还能响应 Ping 命令，验证了网络层的连通性。代码中还实现了多 ARP 表项的管理，支持同时维护多个 IP-MAC 映射。

本次实验的主要技术要点包括：

1. 状态机设计：ARP 表项采用有限状态机设计，通过 FREE、PENDING、RESOLVED 三种状态管理表项生命周期，实现了清晰的转换逻辑和可靠的状态管理
2. 非阻塞地址解析：通过数据包挂起机制，实现了非阻塞式的 ARP 地址解析。当 IP 层需要发送数据但 MAC 地址尚未解析时，数据包被暂时挂起，待 ARP 解析完成后自动发送，避免了数据包丢失

3. 被动学习机制：在接收 ARP 报文和 IP 报文时，自动学习发送方的 IP-MAC 映射并更新 ARP 表，减少了主动 ARP 请求的次数，提高了通信效率
4. 超时重传机制：采用指数退避式的重传策略，设置 1 秒超时时间和 3 次重试上限，在网络不稳定的情况下仍能完成地址解析，同时避免了无限重传
5. 字节序处理：正确处理了网络字节序（大端）和主机字节序的转换，通过 `swap_order16` 宏实现了 16 位整数的字节序交换，确保了跨平台兼容性
6. 校验和计算：实现了标准的互联网校验和算法，用于 IP 头部和 ICMP 报文的完整性验证，提高了协议栈的可靠性
7. 免费 **ARP**：在系统启动时主动发送免费 ARP，通告本机 IP 地址和 MAC 地址，实现了 IP 冲突检测和网络拓扑更新
8. 多表项管理：ARP 表支持 16 个表项的并发管理，能够同时维护多个 IP-MAC 映射，适应了多主机通信的需求

通过本次实验，不仅加深了对 TCP/IP 协议栈的理解，也掌握了网络协议实现的关键技术和设计思想，为后续学习更复杂的网络协议奠定了基础。

5.2 心得感悟

通过本次实验，我深入理解了 ARP 协议在以太网通信中的桥梁作用。从代码层面看，ARP 协议虽然逻辑相对简单，但其状态机（PENDING, RESOLVED）的管理以及与 IP 层的交互细节（如数据包挂起）非常关键。

在实现过程中，对“免费 ARP”的作用有了更直观的认识——它在系统启动时主动通告自身存在，有效避免了 IP 冲突。同时，超时重传机制的实现让我体会到了网络协议在不可靠传输介质上保证可靠性的设计思想。通过结合 Wireshark 抓包调试，我能够清晰地看到数据包的交互流程，这种理论与实践结合的方式极大地巩固了我的计算机网络知识体系。

协议设计思想的思考：

ARP 协议的设计体现了网络协议中的几个重要原则。首先是“简单性原则”，ARP 协议只解决地址映射这一个核心问题，不涉及复杂的路由和转发逻辑，使得实现简单高效。其次是“被动学习”的思想，通过观察网络中的通信自动建立缓存表，减少了不必要的请求。最后是“渐进式可靠性”，通过超时重传机制在不可靠的以太网上实现可靠的地址解析，而不需要复杂的确认应答机制。

调试经验总结：

在实验过程中，我遇到了几个典型的问题。首先是字节序问题，ARP 报文中的 16 位字段需要使用网络字节序（大端），而 x86 架构使用小端字节序，必须进行转换。其次是内存对齐问题，使用 `#pragma pack(1)` 指令确保 ARP 报文结构体紧密排列，避免编译器插入填充字节。最后是状态管理的复杂性，ARP 表项的状态转换需要仔细处理，特别是从 PENDING 状态到 RESOLVED 状态时，必须确保挂起的数据包能够正确发送。

通过 Wireshark 抓包分析，我发现了一个有趣的现象：当连续发送多个 ARP 请求时，如果目标主机不响应，ARP 请求会按照 1 秒的间隔重传 3 次，然后停止。这种设计避免了网络拥塞，也体现了协议设计者的深思熟虑。

改进建议：

基于本次实验的经验，我认为可以从以下几个方面进行改进：

1. **ARP** 缓存淘汰策略：当前实现采用简单的超时淘汰机制，可以引入 LRU（最近最少使用）算法，当 ARP 表满时优先淘汰最久未使用的表项，提高缓存命中率
2. **Gratuitous ARP** 的定期发送：除了启动时发送免费 ARP，可以定期（如每隔几分钟）发送一次，确保网络中的 ARP 缓存保持最新状态
3. **ARP** 欺骗防护：可以增加 ARP 请求的验证机制，例如检查 ARP 响应是否与请求匹配，防止 ARP 欺骗攻击
4. 统计信息收集：增加 ARP 请求成功率、平均响应时间等统计信息，便于性能分析和问题诊断
5. **IPv6** 支持：扩展协议栈以支持 IPv6 的邻居发现协议（NDP），实现下一代网络协议的地址解析功能

通过本次实验，我不仅掌握了 ARP 协议的实现技术，更重要的是学会了如何从协议规范出发，设计并实现一个完整的网络协议模块。这种能力对于后续学习更复杂的网络协议（如 TCP、UDP）以及从事网络相关工作都具有重要意义。