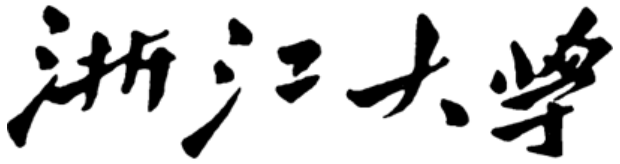

本科实验报告

课程名称:	计算机网络
姓　　名:	刘仁钦
学　　院:	计算机科学与技术学院
专　　业:	计算机科学与技术
学　　号:	3230106230
指导教师:	韩劲松

2025 年 10 月 25 日

目录

浙江大学实验报告

实验项目名称：Lab3 网络接口与 IP 路由器

学生姓名：刘仁钦 专业：计算机科学与技术 学号：3230106230

一、 实验目的

- 学习掌握网络接口的工作原理
- 学习掌握 ARP 地址解析协议相关知识
- 学习掌握 IP 路由的工作原理

二、 实验内容

- 实现 network interface，为每个下一跳地址查找（和缓存）以太网地址，即实现地址解析协议 ARP。
- 实现简易路由器，对于给定的数据包，确认发送接口以及下一跳的 IP 地址。

三、 主要仪器设备

- 联网的 PC 机
- Linux 虚拟机

四、 操作方法与实验步骤

本次实验的核心任务分为两个阶段。首先是实现网络接口（`NetworkInterface`）类，使其能够处理 IP 数据包的发送和接收，并实现 ARP 协议来解析 IP 地址到 MAC 地址的映射。其次是实现路由器（`Router`）类，构建路由表并实现最长前缀匹配算法，以完成 IP 数据包的转发。

1. 实现网络接口类

`NetworkInterface` 类的职责是管理一个网络接口的 IP 地址和 MAC 地址，并负责将 `InternetDatagram`（IP 数据包）封装成 `EthernetFrame`（以太网帧）发送出去，以及解封装接收到的以太网帧。

1.1. 在 `libsponge/network_interface.hh` 中添加必要的成员变量和方法

为了实现 ARP 协议，我们需要在类中添加几个关键的成员变量：

- `_frames_out`：一个队列，用于缓存所有待发送出去的以太网帧。
- `_arp_cache`：ARP 缓存，一个 `std::map`，用于存储 IP 地址到（MAC 地址，剩余超时时间）的映射。这是实现 ARP 高效查询的核心。

- `_pending_datagrams`：一个 `std::map`，用于暂存那些因为目标 MAC 地址未知（ARP 未解析）而无法立即发送的 IP 数据包。键是目标 IP 地址，值是等待该 IP 的包队列。
- `_arp_requests`：一个 `std::map`，用于记录最近发送过的 ARP 请求及其剩余超时时间，以避免在短时间内（5 秒内）对同一 IP 重复发送 ARP 请求。
- `ARP_CACHE_TIMEOUT` 和 `ARP_REQUEST_TIMEOUT`：两个静态常量，分别定义了 ARP 缓存条目的超时时间（30 秒）和 ARP 请求的重传间隔（5 秒）。

具体代码参见附录

1.2. 在 `libsponge/network_interface.cc` 中实现三个方法

我们在 `.cc` 文件中实现了 `NetworkInterface` 的三个核心方法：

1. `send_datagram(dgram, next_hop)`：当上层（IP 层）需要发送一个数据包时，会调用此方法。
    1. 首先，它会查询 `_arp_cache` 寻找下一跳 IP 对应的 MAC 地址。
    2. 如果找到：立即将 IP 数据包封装成一个以太网帧，设置正确的目的 MAC、源 MAC 和类型（IPv4），然后推入 `_frames_out` 队列等待发送。
    3. 如果未找到：说明 MAC 地址未知。此时，数据包不能立即发送，而是被推入 `_pending_datagrams` 队列中暂存。同时，检查 `_arp_requests` 确认 5 秒内是否已发送过对该 IP 的 ARP 请求。如果未发送过，就构造一个 ARP 请求（广播），封装成以太网帧推入 `_frames_out`，并记录请求时间。
2. `recv_frame(frame)`：当网络接口从物理层收到一个以太网帧时，会调用此方法。
    1. 首先检查帧的目的 MAC 地址，如果不是本机 MAC 地址也不是广播地址，则直接丢弃。
    2. 如果是 IPv4 帧：解析 `payload` 为 `InternetDatagram`，并将其返回给上层处理。
    3. 如果是 ARP 帧：解析 `payload` 为 `ARPMessage`。
        1. 首先，无条件学习并缓存发送方的 IP-MAC 映射到 `_arp_cache` 中（设置 30 秒超时）。
        2. 如果是 ARP 请求 且目标 IP 是本机 IP：构造一个 ARP 应答，填入本机 MAC 和 IP，然后将应答帧推入 `_frames_out` 队列。

    3. 如果是 ARP 应答：清除 `_arp_requests` 中对应的记录。然后，检查 `_pending_datagrams` 中是否有等待该 IP 的数据包。如果有，将所有等待的包依次封装成以太网帧（现在我们知道目的 MAC 了）并推入 `_frames_out` 队列。

3. `tick(ms_since_last_tick)`：这是一个定时器方法，由系统周期性调用。
    1. 它负责管理 ARP 缓存和 ARP 请求记录的老化。它会遍历 `_arp_cache` 和 `_arp_requests`，将所有条目的剩余时间减去 `ms_since_last_tick`。如果任何条目的时间减到 0 或以下，就将其从 `map` 中删除。

具体代码参见附录

2. 实现路由类

路由器的核心功能是转发 IP 数据包。它通过查询路由表来决定数据包的下一跳地址和应该从哪个接口发送出去。

2.1. 在 `libsponge/router.hh` 中添加路由表

我们在 `Router` 类中添加了一个 `std::vector<RouteEntry>` 作为路由表。`RouteEntry` 结构体用于存储路由表的每一行，包含四个关键信息：

- `route_prefix`：路由匹配的目标网段前缀。
- `prefix_length`：前缀的长度（0-32 位）。
- `next_hop`：下一跳的 IP 地址。如果 `next_hop` 为空（`std::optional`），表示该路由是直连网络，下一跳就是数据包的最终目的 IP。
- `interface_num`：数据包应该从此索引对应的接口发送出去。

具体代码参见附录

2.2. 在 `libsponge/router.cc` 中实现 `add_route` 和 `route_one_datagram`

`add_route` 方法比较简单，只是将一条新的路由规则（`RouteEntry`）添加到 `_routing_table` 向量中。

`route_one_datagram` 是路由转发的核心逻辑，其步骤如下：

1. 首先检查 IP 数据包头的 `ttl` 字段。如果 `ttl` 小于或等于 1，意味着数据包在转发后即超时，此时应直接丢弃该包，不再转发。否则，将 `ttl` 减 1。

2. 遍历路由表 `_routing_table` 中的所有条目，寻找与数据包目的 IP 地址 `dgram.header().dst` 匹配的最长前缀。
3. 如果未找到匹配：说明路由表中没有到达该目的地的路径，丢弃该数据包。
4. 如果找到匹配：根据匹配到的最佳路由 `best_route`，确定下一跳 `next_hop_addr`（如果路由条目中指定了 `next_hop`，则用它；否则用数据包的原始目的 IP）。
5. 最后，调用该路由指定的 `_interfaces[best_route.interface_num]` 上的 `send_datagram` 方法，将（已更新 TTL 的）数据包和下一跳地址交给网络接口层去处理（后续的 ARP 解析等）。

具体代码参见附录

五、 实验数据记录和处理

1. 实验结果

1.1. 问题一

测试 ARP 协议的运行截图

```bash
ctest -V -R "^arp"
```

运行测试命令，测试结果如 图 1。

```
-> % ctest -V -R "^arp"
UpdateCTestConfiguration  from :/home/cyrus/ZJU-course-Net/zju-comnet-labs/build/DartConfiguration.tcl
Test project /home/cyrus/ZJU-course-Net/zju-comnet-labs/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 32
    Start 32: arp_network_interface

32: Test command: /home/cyrus/ZJU-course-Net/zju-comnet-labs/build/tests/net_interface
32: Working Directory: /home/cyrus/ZJU-course-Net/zju-comnet-labs/build
32: Test timeout computed to be: 10000000
32: DEBUG: Network interface has Ethernet address f6:d5:b9:22:ab:2a and IP address 4.3.2.1
32: DEBUG: Network interface has Ethernet address f6:26:83:ae:3a:60 and IP address 5.5.5.5
32: DEBUG: Network interface has Ethernet address 0a:a9:ec:95:47:74 and IP address 5.5.5.5
32: DEBUG: Network interface has Ethernet address 5e:d8:f6:6a:df:c4 and IP address 1.2.3.4
32: DEBUG: Network interface has Ethernet address ca:45:7e:e4:4f:56 and IP address 4.3.2.1
32: DEBUG: Network interface has Ethernet address 0a:1e:4b:48:7a:5b and IP address 10.0.0.1
1/1 Test #32: arp_network_interface .....   Passed    0.00 sec

The following tests passed:
        arp_network_interface

100% tests passed, 0 tests failed out of 1
```

图 1: 测试 ARP 协议

测试结果显示所有与 ARP 相关的测试（`arp_...`）均已通过。这表明我们的 `NetworkInterface` 能够正确地：

1. 在收到 `send_datagram` 请求时，当 MAC 未知时发送 ARP 请求；
2. 在收到 ARP 请求时，能正确回复 ARP 应答；
3. 在收到 ARP 应答时，能正确更新缓存并发送之前暂存的数据包；
4. 通过 `tick` 方法正确管理 ARP 缓存条目的超时。

1.2. 问题二

运行 `make check_lab1` 命令的测试结果展示

```bash
make check_lab1
```

运行测试命令，测试结果如 图 2。

```
-> % make check_lab1
Testing Lab 1...
Test project /home/cyrus/ZJU-course-Net/zju-comnet-labs/build
    Start 32: arp_network_interface
1/2 Test #32: arp_network_interface .............   Passed    0.00 sec
    Start 33: router_test
2/2 Test #33: router_test .......................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.01 sec
Built target check_lab1
```

图 2: 运行测试

`make check_lab1` 运行了 Lab1 的全部测试用例。测试结果显示全部通过（Passed），这证明我们的路由器实现（包括路由表的最长前缀匹配、TTL 处理）和网络接口实现（ARP 协议、缓存管理）均按预期正常工作。

2. 思考题

2.1. 问题一

通过代码，请描述 network interface 是如何发送一个以太网帧的?

当上层（IP 层）需要发送一个数据包时，会调用 `send_datagram(dgram, next_hop)` 方法。该方法的实现步骤如下：

1. 首先，它会查询 `_arp_cache` 寻找下一跳 IP 对应的 MAC 地址。
2. 如果找到：立即将 IP 数据包封装成一个以太网帧，设置正确的目的 MAC、源 MAC 和类型（IPv4），然后推入 `_frames_out` 队列等待发送。

3. 如果未找到：说明 MAC 地址未知。此时，数据包不能立即发送，而是被推入 `_pending_datagrams` 队列中暂存。同时，检查 `_arp_requests` 确认 5 秒内是否已发送过对该 IP 的 ARP 请求。如果未发送过，就构造一个 ARP 请求（广播），封装成以太网帧推入 `_frames_out`，并记录请求时间。

2.2. 问题二
虽然在此次实验不需要考虑这种情况，但是当 network interface 发送一个 ARP 请求后如果没收到一个应答该怎么解决？请思考。

如果发送 ARP 请求后没有收到应答，这可能意味着目标主机已关机、不在该网络上，或者 ARP 请求（广播）/ARP 应答（单播）在传输过程中丢失了。

标准的处理机制是**重传**和**超时**：

1. 重传机制：网络接口不应只发送一次 ARP 请求就放弃。它应该实现一个重传机制。当发送 ARP 请求时，启动一个定时器,如果定时器超时后仍未收到 ARP 应答，接口应该重新发送一次 ARP 请求，并重置定时器。
2. 限制重传次数：重传不应该是无限的。系统通常会设定一个最大重传次数（例如 3 到 5 次）。
3. 处理最终失败：如果在达到最大重传次数后，仍然没有收到任何应答，网络接口必须假定该 IP 地址在本地链路上是不可达的。
4. 报告错误并丢弃数据包：一旦确定不可达，接口应该丢弃所有在 `_pending_datagrams` 队列中等待该 IP 地址解析的数据包。向上层（IP 层）报告一个“主机不可达”（Host Unreachable）的错误。这通常会触发 IP 层向原始发送方返回一个 ICMP 错误消息，最终通知到传输层（如 TCP），导致连接超时或失败。

2.3. 问题三
请描述一下你为了记录路由表所建立的数据结构？为什么？

为了记录路由表，我使用了一个 `std::vector<RouteEntry>`（C++标准库中的动态数组）作为核心数据结构。

```cpp
//libsponge/router.hh
struct RouteEntry {
    uint32_t route_prefix;      // 路由前缀
    uint8_t prefix_length;     // 前缀长度
    std::optional<Address> next_hop; // 下一跳地址
    size_t interface_num;      // 对应的接口索引
```

```
7 };
8
9 std::vector<RouteEntry> _routing_table{};
```

`_routing_table` 存储的是 `RouteEntry` 结构体。每个 `RouteEntry` 实例就代表路由表中的“一行”，包含了路由决策所需的全部信息：要匹配的网段前缀、前缀长度、下一跳 IP 地址以及数据包应从哪个接口发出。

选择 `std::vector` 这种数据结构的原因如下：

1. 实现简单直观，`add_route`（添加路由）操作可以非常简单地通过 `_routing_table.push_back()`（在末尾追加）来实现。
2. 易于实现最长前缀匹配：路由转发的核心是“最长前缀匹配”算法。使用 `vector`，可以线性扫描每一个路由表项，并通过位运算让每个条目的检查非常快，虽然复杂度是$O(N)$，但是常数是很小的，在实验要求的规模下可以接受，不必使用 Trie 树来实现。

六、 讨论、心得

通过本次实验，我将计算机网络课程中关于数据链路层和网络层的抽象理论知识，转化为了具体的代码实践。在实现 `NetworkInterface` 的过程中，我亲手实现了 ARP 协议。通过管理 ARP 缓存、处理超时（`tick`）以及暂存等待解析的数据包，我深刻理解了 IP 地址是如何在数据链路层被动态解析为 MAC 地址的。在实现 `Router` 时，核心是实现了“最长前缀匹配”算法和 TTL 的递减。这让我非常直观地看到了路由器是如何根据路由表，从多条规则中找出最佳路径并转发数据包的。本次实验让我对数据链路层（L2）和网络层（L3）如何协同工作，以及数据包如何在网络中被转发有了更扎实和深刻的认识。

七、 附录

1. `libsponge/network_interface.hh`

libsponge/network_interface.hh cpp

```
1 #ifndef SPONGE_LIBSPONGE_NETWORK_INTERFACE_HH
2 #define SPONGE_LIBSPONGE_NETWORK_INTERFACE_HH
3
4 #include "ethernet_frame.hh"
5 #include "tcp_over_ip.hh"
6 #include "tun.hh"
7
```

```
8  #include <map>
9  #include <optional>
10 #include <queue>
11
12 //! \brief A "network interface" that connects IP (the internet layer, or
   network layer)
13 //! with Ethernet (the network access layer, or link layer).
14
15 //! This module is the lowest layer of a TCP/IP stack
16 //! (connecting IP with the lower-layer network protocol,
17 //! e.g. Ethernet). But the same module is also used repeatedly
18 //! as part of a router: a router generally has many network
19 //! interfaces, and the router's job is to route Internet datagrams
20 //! between the different interfaces.
21
22 //! The network interface translates datagrams (coming from the
23 //! "customer," e.g. a TCP/IP stack or router) into Ethernet
24 //! frames. To fill in the Ethernet destination address, it looks up
25 //! the Ethernet address of the next IP hop of each datagram, making
26 //! requests with the [Address Resolution Protocol](\ref rfc::rfc826).
27 //! In the opposite direction, the network interface accepts Ethernet
28 //! frames, checks if they are intended for it, and if so, processes
29 //! the the payload depending on its type. If it's an IPv4 datagram,
30 //! the network interface passes it up the stack. If it's an ARP
31 //! request or reply, the network interface processes the frame
32 //! and learns or replies as necessary.
33 class NetworkInterface {
34   private:
35     //! Ethernet (known as hardware, network-access-layer, or link-layer)
       address of the interface
36     EthernetAddress _ethernet_address;
37
38     //! IP (known as internet-layer or network-layer) address of the
       interface
39     Address _ip_address;
40
41     //! outbound queue of Ethernet frames that the NetworkInterface wants
       sent
42     std::queue<EthernetFrame> _frames_out{};
43
44     //! ARP cache: maps IP address to Ethernet address and remaining time
       (in milliseconds)
```

```
45      std::map<uint32_t, std::pair<EthernetAddress, size_t>> _arp_cache{};
46
47      //! Pending datagrams waiting for ARP resolution: maps IP address to
        queue of datagrams
48      std::map<uint32_t, std::queue<InternetDatagram>>
        _pending_datagrams{};
49
50      //! ARP requests sent: maps IP address to remaining time before
        retransmission (in milliseconds)
51      std::map<uint32_t, size_t> _arp_requests{};
52
53      //! ARP cache timeout in milliseconds (30 seconds)
54      static constexpr size_t ARP_CACHE_TIMEOUT = 30000;
55
56      //! ARP request timeout in milliseconds (5 seconds)
57      static constexpr size_t ARP_REQUEST_TIMEOUT = 5000;
58
59    public:
60      //! \brief Construct a network interface with given Ethernet
        (network-access-layer) and IP (internet-layer) addresses
61      NetworkInterface(const EthernetAddress &ethernet_address, const
        Address &ip_address);
62
63      //! \brief Access queue of Ethernet frames awaiting transmission
64      std::queue<EthernetFrame> &frames_out() { return _frames_out; }
65
66      //! \brief Sends an IPv4 datagram, encapsulated in an Ethernet frame
        (if it knows the Ethernet destination address).
67
68      //! Will need to use [ARP](\ref rfc::rfc826) to look up the Ethernet
        destination address for the next hop
69      //! ("Sending" is accomplished by pushing the frame onto the
        frames_out queue.)
70      void send_datagram(const InternetDatagram &dgram, const Address
        &next_hop);
71
72      //! \brief Receives an Ethernet frame and responds appropriately.
73
74      //! If type is IPv4, returns the datagram.
75      //! If type is ARP request, learn a mapping from the "sender" fields,
        and send an ARP reply.
76      //! If type is ARP reply, learn a mapping from the "sender" fields.
```

```
77      std::optional<InternetDatagram> recv_frame(const EthernetFrame
        &frame);
78
79      //! \brief Called periodically when time elapses
80      void tick(const size_t ms_since_last_tick);
81  };
82
83  #endif  // SPONGE_LIBSPONGE_NETWORK_INTERFACE_HH
```

2. **libsponge/network_interface.cc**

libsponge/network_interface.cc cpp

```
1   #include "network_interface.hh"
2
3   #include "arp_message.hh"
4   #include "ethernet_frame.hh"
5
6   #include <iostream>
7
8   using namespace std;
9
10  //! \param[in] ethernet_address Ethernet (what ARP calls "hardware")
    address of the interface
11  //! \param[in] ip_address IP (what ARP calls "protocol") address of the
    interface
12  NetworkInterface::NetworkInterface(const EthernetAddress
    &ethernet_address, const Address &ip_address)
13      : _ethernet_address(ethernet_address), _ip_address(ip_address) {
14      cerr << "DEBUG: Network interface has Ethernet address " <<
        to_string(_ethernet_address) << " and IP address "
15           << ip_address.ip() << "\n";
16  }
17
18  //! \param[in] dgram the IPv4 datagram to be sent
19  //! \param[in] next_hop the IP address of the interface to send it to
    (typically a router or default gateway, but may also be another host if
    directly connected to the same network as the destination)
20  //! (Note: the Address type can be converted to a uint32_t (raw 32-bit
    IP address) with the Address::ipv4_numeric() method.)
21  void NetworkInterface::send_datagram(const InternetDatagram &dgram,
    const Address &next_hop) {
22      // convert IP address of next hop to raw 32-bit representation (used
        in ARP header)
```

```
23      const uint32_t next_hop_ip = next_hop.ipv4_numeric();
24
25      // Check if the Ethernet address is already in the ARP cache
26      auto arp_it = _arp_cache.find(next_hop_ip);
27      if (arp_it != _arp_cache.end()) {
28          // Ethernet address is known, send the frame immediately
29          EthernetFrame frame;
30          frame.header().dst = arp_it->second.first;        // Destination
            MAC address
31          frame.header().src = _ethernet_address;           // Source MAC
            address
32          frame.header().type = EthernetHeader::TYPE_IPv4;  // IPv4 type
33          frame.payload() = dgram.serialize();              // Serialize
            the datagram as payload
34          _frames_out.push(frame);
35      } else {
36          // Ethernet address is unknown, queue the datagram
37          _pending_datagrams[next_hop_ip].push(dgram);
38
39          // Check if we've already sent an ARP request for this IP
            recently
40          auto arp_req_it = _arp_requests.find(next_hop_ip);
41          if (arp_req_it == _arp_requests.end()) {
42              // No recent ARP request, send a new one
43              ARPMessage arp_request;
44              arp_request.opcode = ARPMessage::OPCODE_REQUEST;
45              arp_request.sender_ethernet_address = _ethernet_address;
46              arp_request.sender_ip_address = _ip_address.ipv4_numeric();
47              arp_request.target_ethernet_address = {0, 0, 0, 0, 0,
                0};  // Unknown
48              arp_request.target_ip_address = next_hop_ip;
49
50              // Encapsulate ARP request in Ethernet frame and broadcast
51              EthernetFrame frame;
52              frame.header().dst = ETHERNET_BROADCAST;  // Broadcast
                address
53              frame.header().src = _ethernet_address;
54              frame.header().type = EthernetHeader::TYPE_ARP;
55              frame.payload() = BufferList(arp_request.serialize());
56              _frames_out.push(frame);
57
58              // Record the ARP request time
```

```
59              _arp_requests[next_hop_ip] = ARP_REQUEST_TIMEOUT;
60          }
61      }
62  }
63
64  //! \param[in] frame the incoming Ethernet frame
65  optional<InternetDatagram> NetworkInterface::recv_frame(const
    EthernetFrame &frame) {
66      // Check if the frame is intended for this interface (or is a
        broadcast)
67      const EthernetAddress &dst = frame.header().dst;
68      if (dst != _ethernet_address && dst != ETHERNET_BROADCAST) {
69          // Frame is not for us, discard it
70          return {};
71      }
72
73      // Handle IPv4 datagram
74      if (frame.header().type == EthernetHeader::TYPE_IPv4) {
75          InternetDatagram dgram;
76          if (dgram.parse(frame.payload()) == ParseResult::NoError) {
77              return dgram;
78          }
79      }
80      // Handle ARP message
81      else if (frame.header().type == EthernetHeader::TYPE_ARP) {
82          ARPMessage arp_msg;
83          if (arp_msg.parse(frame.payload()) == ParseResult::NoError) {
84              // Learn the mapping from sender IP to sender Ethernet
                address
85              const uint32_t sender_ip = arp_msg.sender_ip_address;
86              const EthernetAddress sender_eth =
                arp_msg.sender_ethernet_address;
87
88              // Add to ARP cache with 30-second timeout
89              _arp_cache[sender_ip] = {sender_eth, ARP_CACHE_TIMEOUT};
90
91              // If this is an ARP request for our IP address, send a
                reply
92              if (arp_msg.opcode == ARPMessage::OPCODE_REQUEST &&
93                  arp_msg.target_ip_address == _ip_address.ipv4_numeric())
                    {
94                  // Create ARP reply
```

```
95                  ARPMessage arp_reply;
96                  arp_reply.opcode = ARPMessage::OPCODE_REPLY;
97                  arp_reply.sender_ethernet_address = _ethernet_address;
98                  arp_reply.sender_ip_address =
                    _ip_address.ipv4_numeric();
99                  arp_reply.target_ethernet_address = sender_eth;
100                 arp_reply.target_ip_address = sender_ip;
101
102                 // Encapsulate in Ethernet frame
103                 EthernetFrame reply_frame;
104                 reply_frame.header().dst = sender_eth;
105                 reply_frame.header().src = _ethernet_address;
106                 reply_frame.header().type = EthernetHeader::TYPE_ARP;
107                 reply_frame.payload() =
                    BufferList(arp_reply.serialize());
108                 _frames_out.push(reply_frame);
109             }
110             // If this is an ARP reply, send any pending datagrams
111             else if (arp_msg.opcode == ARPMessage::OPCODE_REPLY) {
112                 // Remove the ARP request record
113                 _arp_requests.erase(sender_ip);
114
115                 // Send all pending datagrams for this IP
116                 auto pending_it = _pending_datagrams.find(sender_ip);
117                 if (pending_it != _pending_datagrams.end()) {
118                     while (!pending_it->second.empty()) {
119                         const InternetDatagram &dgram = pending_it-
                            >second.front();
120
121                         // Create and send the frame
122                         EthernetFrame eth_frame;
123                         eth_frame.header().dst = sender_eth;
124                         eth_frame.header().src = _ethernet_address;
125                         eth_frame.header().type =
                            EthernetHeader::TYPE_IPv4;
126                         eth_frame.payload() = dgram.serialize();
127                         _frames_out.push(eth_frame);
128
129                         pending_it->second.pop();
130                     }
131                     // Remove the empty queue
132                     _pending_datagrams.erase(pending_it);
```

```
133                 }
134             }
135         }
136     }
137
138     return {};
139 }
140
141 //! \param[in] ms_since_last_tick the number of milliseconds since the last call to this method
142 void NetworkInterface::tick(const size_t ms_since_last_tick) {
143     // Update ARP cache entries and remove expired ones
144     for (auto it = _arp_cache.begin(); it != _arp_cache.end();) {
145         if (it->second.second <= ms_since_last_tick) {
146             // Entry has expired, remove it
147             it = _arp_cache.erase(it);
148         } else {
149             // Decrement remaining time
150             it->second.second -= ms_since_last_tick;
151             ++it;
152         }
153     }
154
155     // Update ARP request timers and remove expired ones
156     for (auto it = _arp_requests.begin(); it != _arp_requests.end();) {
157         if (it->second <= ms_since_last_tick) {
158             // ARP request has expired, remove it (will be resent on next send_datagram)
159             it = _arp_requests.erase(it);
160         } else {
161             // Decrement remaining time
162             it->second -= ms_since_last_tick;
163             ++it;
164         }
165     }
166 }
```

3. `libsponge/router.hh`

libsponge/router.hh cpp

```
1 #ifndef SPONGE_LIBSPONGE_ROUTER_HH
2 #define SPONGE_LIBSPONGE_ROUTER_HH
```

```
3
4  #include "network_interface.hh"
5
6  #include <optional>
7  #include <queue>
8
9  //! \brief A wrapper for NetworkInterface that makes the host-side
10 //! interface asynchronous: instead of returning received datagrams
11 //! immediately (from the `recv_frame` method), it stores them for
12 //! later retrieval. Otherwise, behaves identically to the underlying
13 //! implementation of NetworkInterface.
14 class AsyncNetworkInterface : public NetworkInterface {
15     std::queue<InternetDatagram> _datagrams_out{};
16
17   public:
18     using NetworkInterface::NetworkInterface;
19
20     //! Construct from a NetworkInterface
21     AsyncNetworkInterface(NetworkInterface &&interface) :
       NetworkInterface(interface) {}
22
23     //! \brief Receives and Ethernet frame and responds appropriately.
24
25     //! - If type is IPv4, pushes to the `datagrams_out` queue for later
       retrieval by the owner.
26     //! - If type is ARP request, learn a mapping from the "sender"
       fields, and send an ARP reply.
27     //! - If type is ARP reply, learn a mapping from the "target" fields.
28     //!
29     //! \param[in] frame the incoming Ethernet frame
30     void recv_frame(const EthernetFrame &frame) {
31         auto optional_dgram = NetworkInterface::recv_frame(frame);
32         if (optional_dgram.has_value()) {
33             _datagrams_out.push(std::move(optional_dgram.value()));
34         }
35     };
36
37     //! Access queue of Internet datagrams that have been received
38     std::queue<InternetDatagram> &datagrams_out() { return
       _datagrams_out; }
39 };
40
```

```
41 //! \brief A router that has multiple network interfaces and
42 //! performs longest-prefix-match routing between them.
43 class Router {
44     //! The router's collection of network interfaces
45     std::vector<AsyncNetworkInterface> _interfaces{};
46
47     //! \brief A single entry in the routing table
48     struct RouteEntry {
49         uint32_t route_prefix;                //!< The route prefix to
            match
50         uint8_t prefix_length;                //!< Number of high-order
            bits to match
51         std::optional<Address> next_hop;      //!< Next hop address
            (empty if directly attached)
52         size_t interface_num;                 //!< Index of the
            interface to send out on
53     };
54
55     //! The routing table
56     std::vector<RouteEntry> _routing_table{};
57
58     //! Send a single datagram from the appropriate outbound interface to
       the next hop,
59     //! as specified by the route with the longest prefix_length that
       matches the
60     //! datagram's destination address.
61     void route_one_datagram(InternetDatagram &dgram);
62
63   public:
64     //! Add an interface to the router
65     //! \param[in] interface an already-constructed network interface
66     //! \returns The index of the interface after it has been added to
       the router
67     size_t add_interface(AsyncNetworkInterface &&interface) {
68         _interfaces.push_back(std::move(interface));
69         return _interfaces.size() - 1;
70     }
71
72     //! Access an interface by index
73     AsyncNetworkInterface &interface(const size_t N) { return
       _interfaces.at(N); }
74
```

```cpp
    //! Add a route (a forwarding rule)
    void add_route(const uint32_t route_prefix,
                   const uint8_t prefix_length,
                   const std::optional<Address> next_hop,
                   const size_t interface_num);

    //! Route packets between the interfaces
    void route();
};

#endif  // SPONGE_LIBSPONGE_ROUTER_HH
```

4. `libsponge/router.cc`

libsponge/router.cc cpp

```cpp
#include "router.hh"

#include <iostream>

using namespace std;

//! \param[in] route_prefix The "up-to-32-bit" IPv4 address prefix to match the datagram's destination address against
//! \param[in] prefix_length For this route to be applicable, how many high-order (most-significant) bits of the route_prefix will need to match the corresponding bits of the datagram's destination address?
//! \param[in] next_hop The IP address of the next hop. Will be empty if the network is directly attached to the router (in which case, the next hop address should be the datagram's final destination).
//! \param[in] interface_num The index of the interface to send the datagram out on.
void Router::add_route(const uint32_t route_prefix,
                       const uint8_t prefix_length,
                       const optional<Address> next_hop,
                       const size_t interface_num) {
    cerr << "DEBUG: adding route " <<
    Address::from_ipv4_numeric(route_prefix).ip() << "/" <<
    int(prefix_length)
         << " => " << (next_hop.has_value() ? next_hop->ip() :
         "(direct)") << " on interface " << interface_num << "\n";

    // Add the route entry to the routing table
```

```
    _routing_table.push_back({route_prefix, prefix_length, next_hop, interface_num});
}

//! \param[in] dgram The datagram to be routed
void Router::route_one_datagram(InternetDatagram &dgram) {
    // Check if TTL is valid (must be > 1 to forward after decrement)
    if (dgram.header().ttl <= 1) {
        return;  // Discard the datagram (TTL expired or will expire)
    }

    // Decrement TTL
    dgram.header().ttl--;

    // Find the longest prefix match in the routing table
    const uint32_t dst_ip = dgram.header().dst;
    int best_match_idx = -1;
    uint8_t longest_prefix = 0;

    for (size_t i = 0; i < _routing_table.size(); i++) {
        const auto &route = _routing_table[i];
        const uint8_t prefix_len = route.prefix_length;

        // Create a mask for the prefix
        uint32_t mask = 0;
        if (prefix_len > 0) {
            if (prefix_len == 32) {
                mask = 0xFFFFFFFF;
            } else {
                mask = ~((1U << (32 - prefix_len)) - 1);
            }
        }

        // Check if the destination IP matches this route's prefix
        if ((dst_ip & mask) == (route.route_prefix & mask)) {
            // This route matches; check if it's the longest prefix so far
            if (prefix_len >= longest_prefix) {
                longest_prefix = prefix_len;
                best_match_idx = i;
            }
        }
```

```
59      }
60
61      // If no matching route found, discard the datagram
62      if (best_match_idx == -1) {
63          return;
64      }
65
66      // Get the best matching route
67      const auto &best_route = _routing_table[best_match_idx];
68
69      // Determine the next hop address
70      Address next_hop_addr =
        best_route.next_hop.value_or(Address::from_ipv4_numeric(dst_ip));
71
72      // Send the datagram out on the appropriate interface
73      _interfaces[best_route.interface_num].send_datagram(dgram,
        next_hop_addr);
74  }
75
76  void Router::route() {
77      // Go through all the interfaces, and route every incoming datagram
        to its proper outgoing interface.
78      for (auto &interface : _interfaces) {
79          auto &queue = interface.datagrams_out();
80          while (not queue.empty()) {
81              route_one_datagram(queue.front());
82              queue.pop();
83          }
84      }
85  }
```