计网lab5

实验结果：

```
wsll@wsll-virtual-machine:~/桌面/Conputer_Network_code/minnow$ cmake --build build --target check5
Test project /home/wsll/桌面/Conputer_Network_code/minnow/build
    Start  1: compile with bug-checkers
1/2 Test  #1: compile with bug-checkers ........   Passed    0.71 sec
    Start 35: net_interface
2/2 Test #35: net_interface ....................   Passed    0.07 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.79 sec
Built target check5
```

实现细节：

头文件

```
struct ArpEntry
{
  EthernetAddress ethernet_address;
  uint64_t timestamp;
};

std::unordered_map<uint32_t,ArpEntry> arp_table_ {};
std::unordered_map<uint32_t,uint64_t> arp_request_timestamps_{};//ip对应的最近一次arp请求的时间戳
std::unordered_map<uint32_t,std::queue<EthernetFrame>> wait_queue_{}; //这个队列维护了对某个ip发送过的arp请求的等待队列。如果已经得到了arp回复，就把其从中移除

inline ARPMessage build_ARPMessage(uint16_t opcode,const EthernetAddress& target_ethernet_address,uint32_t target_ip_address);
inline EthernetFrame build_Frame(EthernetAddress dst,EthernetAddress src,uint16_t type,const std::vector<std::string>& payload);
uint64_t timer_{};
```

添加了以上的数据结构。其中ArpEntry是记录了某个mac帧最近一次更新的时间戳。其余可见注释。

构建arp数据和mac帧

```
inline ARPMessage NetworkInterface::build_ARPMessage(uint16_t opcode,const EthernetAddress& target_ethernet_address,uint32_t target_ip_address )
{
  ARPMessage arp_request;
  arp_request.opcode = opcode;
  arp_request.sender_ethernet_address = ethernet_address_;
  arp_request.sender_ip_address = ip_address_.ipv4_numeric();
  arp_request.target_ethernet_address = target_ethernet_address;
  arp_request.target_ip_address = target_ip_address;
  return arp_request;
}
inline EthernetFrame NetworkInterface::build_Frame(EthernetAddress dst,EthernetAddress src,uint16_t type,const std::vector<std::string>& payload)
{
  EthernetFrame res;
  res.header.dst = dst;
  res.header.src = src;
  res.header.type = type;
  res.payload = payload;
  return res;
}
```

按照传进的参数构建。

send_datagram

```cpp
//! \param[in] dgram the IPv4 datagram to be sent
//! \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) Note: the Address type
//! can be converted to a uint32_t (raw 32-bit IP address) by using the Address::ipv4_numeric() method.
void NetworkInterface::send_datagram( const InternetDatagram& dgram, const Address& next_hop )
{
  uint32_t next_hop_ip = next_hop.ipv4_numeric();
  auto arp_entry_it = arp_table_.find( next_hop_ip );

  if ( arp_entry_it != arp_table_.end() ) { // 说明在arp table里找到了，那么直接发就行
    EthernetFrame frame = build_Frame(arp_entry_it->second.ethernet_address,ethernet_address_,EthernetHeader::TYPE_IPv4,serialize(dgram));
    transmit( frame );
  } else {
    EthernetFrame wait_frame = build_Frame(ETHERNET_BROADCAST,ethernet_address_,EthernetHeader::TYPE_IPv4,serialize(dgram));// 得去广播找
    wait_queue_[next_hop_ip].push( wait_frame ); // 放到等待队列去

    // ip没有对应的arp或者已经超时了，那么就得更新了
    auto it = arp_request_timestamps_.find( next_hop_ip );
    if ( it == arp_request_timestamps_.end() || timer_ - it->second >= 5000 ) {
      ARPMessage arp_request = build_ARPMessage(ARPMessage::OPCODE_REQUEST,{},next_hop_ip);
      EthernetFrame arp_frame = build_Frame(ETHERNET_BROADCAST,ethernet_address_,EthernetHeader::TYPE_ARP,serialize(arp_request));
      transmit( arp_frame );
      arp_request_timestamps_[next_hop_ip] = timer_;
    }
  }
}
```

先在arp表中查询，如果查到了，那么就构建mac帧发送过去。如果没有，那么就需要构建arp去找。先放进等待队列里，然后根据等待队列的状态去决定是否需要发送。

recv_frame

```cpp
//! \param[in] frame the incoming Ethernet frame
void NetworkInterface::recv_frame( const EthernetFrame& frame )
{
  // 不是发给我的
  if ( frame.header.dst != ethernet_address_ && frame.header.dst != ETHERNET_BROADCAST ) {
    return;
  }

  if ( frame.header.type == EthernetHeader::TYPE_IPv4 ) {//是ip请求就交给上层
    InternetDatagram datagram;
    if ( parse( datagram, frame.payload ) ) {
      datagrams_received_.push( datagram );
    }
  }
  else if ( frame.header.type == EthernetHeader::TYPE_ARP )
  {
    ARPMessage arp_msg;
    if ( parse( arp_msg, frame.payload ) ) {
      ArpEntry& entry = arp_table_[arp_msg.sender_ip_address];
      entry.ethernet_address = arp_msg.sender_ethernet_address;
      entry.timestamp = timer_;
      // 发送arp回复
      if ( arp_msg.opcode == ARPMessage::OPCODE_REQUEST
           && arp_msg.target_ip_address == ip_address_.ipv4_numeric() ) {
        ARPMessage arp_reply = build_ARPMessage(ARPMessage::OPCODE_REPLY,arp_msg.sender_ethernet_address,arp_msg.sender_ip_address);
        EthernetFrame reply_frame = build_Frame(arp_msg.sender_ethernet_address,ethernet_address_,EthernetHeader::TYPE_ARP,serialize(arp_reply));
        transmit( reply_frame );
      }

      // 检查这个ip对应的等待队列中是否有针对该 IP 的帧
      auto wait_queue_it = wait_queue_.find( arp_msg.sender_ip_address );
      if ( wait_queue_it != wait_queue_.end() ) { // 如果有的话，那么得把这个ip对应的等待队列中的arp全部处理掉
        std::queue<EthernetFrame>& frames = wait_queue_it->second;
        while ( !frames.empty() ) {
          EthernetFrame& pending_frame = frames.front();
          pending_frame.header.dst = arp_msg.sender_ethernet_address;
          transmit( pending_frame );
          frames.pop();
        }
        wait_queue_.erase( wait_queue_it );
      }
    }
  }
  else
  {
    assert( false && "Unsupported Ethernet frame type" );
  }
}
```

如果自己不是目标，那么直接丢弃。如果是ip请求那么就传递给上层。如果是arp，那么发送arp回复。由于arp报文中我们可以得知发送者的ip和mac帧，那么就需要更新waitqueue。

tick

```
//! \param[in] ms_since_last_tick the number of milliseconds since the last call to this method
void NetworkInterface::tick( const size_t ms_since_last_tick )
{
  timer_ += ms_since_last_tick;
  const uint64_t ARP_ENTRY_TTL = 30000;
  //删掉过期的arp表项
  for ( auto it = arp_table_.begin(); it != arp_table_.end(); ) {
    if ( timer_ - it->second.timestamp > ARP_ENTRY_TTL ) {
      it = arp_table_.erase( it );
    } else {
      ++it;
    }
  }
}
```

计时器更新，同时更新arp表。