CS305 Programming Assignment1

Code Interpretation

- Basic Design
- DNS Parsing
- [Most Difficult Part] DNS Encoding
- DNS Server
  - Basic design
  - Handling the case where the Answer section contains only CNAME records

Testing Result

> Name：黄政东
>
> SID：12212230

Code Interpretation

Basic Design

This project is mainly based on the provided Python template to implement specific functions, primarily divided into two main parts: DNS data processing and DNS server. At the same time, reusable code is extracted in the DNS data processing logic as static functions under the py files.

DNS Parsing

The DNS parsing part mainly implements the class methods from_wire of DNSHeader, DNSQuestion, DNSRR, and DNSMessage classes. The from_wire methods of the first three classes will read the byte stream data and the current data offset idx (default idx is 0 for testing purposes). Then, this function will parse the data and return an object containing the data. The from_wire method of the DNSMessage class will call the from_wire methods of the first three classes to progressively parse the DNS message and create a complete message object.

[Most Difficult Part] DNS Encoding

To meet the requirements of sending specific queries by the DNS server, this project also extends the related modules of DNS data processing to support re-encoding DNS message objects into byte streams. It adopts a compression format for domain names to reduce the size of the byte stream. The implementation of this feature is the most challenging part of the entire project.

To implement the DNS encoding feature, the project adds a relatively unified encode_bytes method to the DNSHeader, DNSQuestion, DNSRR, and DNSMessage classes. External users can encode the data inside the objects into byte streams by calling this method.

The implementation of the encode_bytes method of the DNSMessage class is as follows:

DNSMessage Python

```python
def encode_bytes(self) -> bytes:
    res = self.header.encoder_bytes()
    domain_dict = {}
    seg, domain_dict = self.question.encode_bytes(len(res), domain_dict)
    res += seg
    for rr in self.answer + self.authority + self.additional:
        seg, domain_dict = rr.encode_bytes(len(res), domain_dict)
        res += seg
    return res
```

The method first calls the self.header.encoder_bytes() method to convert the DNSHeader part into a byte stream. Then it creates a domain_dict = {} object to store domain segments and their first occurrence positions. Subsequently, it iteratively calls the encode_bytes method of DNSQuestion objects and DNSRR objects. This method takes two parameters: the offset of the current data segment's starting position in the final byte stream and the domain name preceding this data segment along with the offset. The method will then return the corresponding byte stream of this data segment along with the updated domain-name offset dictionary. By gradually concatenating byte stream segments and updating the domain-name offset dictionary, this method will ultimately obtain and return the byte stream of the DNS message body.

The encode_bytes method of the DNSHeader class is implemented as description. The method's logic is relatively simple, converting attributes into byte streams and incrementally concatenating them based on the DNS header format. Finally, the method obtains and returns the byte stream corresponding to the data segment.

DNSHeader Python

```python
def encoder_bytes(self) -> bytes:
    id_bytes = self.id.to_bytes(2, byteorder='big')
    qdcount_bytes = self.qdcount.to_bytes(2, byteorder='big')
    ancount_bytes = self.ancount.to_bytes(2, byteorder='big')
    nscount_bytes = self.nscount.to_bytes(2, byteorder='big')
    arcount_bytes = self.arcount.to_bytes(2, byteorder='big')
    return (id_bytes + self.flag + qdcount_bytes +
            ancount_bytes + nscount_bytes + arcount_bytes)
```

The encode_bytes method for DNSRR and DNSQuestion is quite similar. Here, we present the encode_bytes method for DNSQuestion.

DNSQuestion Python

```python
def encode_bytes(self, previous_len: int, previous_dict: Dict[str, int]) \
-> Tuple[bytes, Dict[str, int]]:
    qname_bytes, _, previous_dict = encode_dns_name(self.qname, previous_le
n,
                                                    previous_dict)
    qtype_bytes = self.qtype.to_bytes(2, byteorder='big')
    qclass_bytes = self.qclass.to_bytes(2, byteorder='big')
    return qname_bytes + qtype_bytes + qclass_bytes, previous_dict
```

The method similarly parses the data step by step according to the format and ultimately concatenates the corresponding byte stream. In the domain name parsing part, the encode_dns_name static method is called for parsing. This method is an important implementation for domain name compression encoding. The code is as follows:

encode_dns_name Python

```python
def encode_dns_name(name: str, prev_len: int, prev_dict: Dict[str, int]) -> Tuple[bytes, int, Dict[str, int]]:
    if name == 'root':
        return b'\x00', 1, prev_dict
    name = '.' + name
    res = b''
    i = 0
    while i < len(name):
        if name[i] != '.':
            i += 1
            continue
        key = name[i + 1:]
        if key in prev_dict:  # 指针模式
            val = prev_dict[key] + (192 << 8)
            res += val.to_bytes(2, byteorder='big')
            return res, len(res), prev_dict
        # 普通模式
        i += 1
        begin = i
        while i < len(name) and name[i] != '.':
            # 找到chunck的len,注意这个chunck找到i停留在.上
            i += 1
        seg = name[begin:i]
        a = len(res)
        res += len(seg).to_bytes(1, byteorder='big')
        res += seg.encode('utf-8')
        # assert a + len(seg) + 1 == len(res)
        # 添加name[begin:]到prev_dict中,其位置为begin-1+prev_len
        prev_dict[name[begin:]] = begin - 1 + prev_len
    return res + b'\x00', len(res) + 1, prev_dict
```

In the case of domain names, there are three scenarios: the first one is the root domain, where the domain name needs to be encoded as b'\x00'. As the byte size of pointer-type domain name encoding is larger than encoding the root server domain name, when the root domain name repeats, there is no need to use pointer-type encoding to reduce the length of the byte stream, and thus the prev_dict is not updated.

Subsequently, the code iterates through the domain name and separates it using the '.' symbol. For each segment of the domain name that needs to be encoded in the current loop (e.g., in www.baidu.com, there are three segments: www.baidu.com, baidu.com, .com), it checks if the segment has appeared previously in prev_dict. If it exists, a pointer-type encoding is added in the domain name byte stream

pointing to the offset stored in the dictionary, and the final result is returned. If the segment does not exist, it encodes one part of the domain name (e.g., encoding www. in www.baidu.com and the remaining domain segment baidu.com), adds this domain segment and the byte stream starting position offset to the dictionary, and continues looping to encode the remaining domain segments until completion.

The above code implements the compression process of domain name encoding, effectively reducing the length of the final byte stream of the DNS message.

In the part related to DNS parsing and encoding, correctly implementing the parsing process for special data such as domain name parsing and encoding, IPv6 abbreviation and full representation conversion is crucial. However, It's a bit challenging for verification&debugging due to the difficulty in readability of byte streams. In this project, unit tests using unittest are conducted on the relevant interface functions to ensure the correctness and reliability of the implementation. Below is a snippet of the code for this module.

```python
class TestStaticFunc(unittest.TestCase):
    def test_encode_and_decode(self):
        b, i, d = encode_dns_name('baidu.com', 0, {})
        b2, i2, d = encode_dns_name('www.baidu.com', i, d)
        b += b2
        str, _ = parse_dns_name(b, i)
        assert str == 'www.baidu.com'

    def test_ipv6_shorten(self):
        assert shorten_ipv6("0:0:0:0:0:0:0:0") == '::'
        assert shorten_ipv6("0:0:0:0:1:0:0:1") == '::1:0:0:1'
        assert shorten_ipv6("0:0:0:0:1:0:0:0") == '::1:0:0:0'
        assert shorten_ipv6("1:0:0:0:1:0:0:0") == '1::1:0:0:0' or shorten_ipv6("1:0:0:0:1:0:0:0") == '1:0:0:0:1::'
        assert shorten_ipv6("1:0:0:0:1:1:1:0") == '1::1:1:1:0'
        assert shorten_ipv6("1:1:1:1:1:0:0:0") == '1:1:1:1:1::'
        assert shorten_ipv6("1:0:0:0:1:0:0:114") == '1::1:0:0:114'

    def test_ipv6_extend(self):
        assert extend_ipv6('1::1') == '1:0:0:0:0:0:0:1'
        assert extend_ipv6('::') == '0:0:0:0:0:0:0:0'
        assert extend_ipv6('1::') == '1:0:0:0:0:0:0:0'
        assert extend_ipv6('::1') == '0:0:0:0:0:0:0:1'
        assert extend_ipv6('1::1:0:0:0') == '1:0:0:0:1:0:0:0'
        assert extend_ipv6('1:0:0:0:1::') == '1:0:0:0:1:0:0:0'
        assert extend_ipv6('1:1:1:1:1::') == '1:1:1:1:1:0:0:0'
        assert extend_ipv6('1::1:0:0:114') == '1:0:0:0:1:0:0:114'

    def test_message(self):
        dns_response_bytes = read_bytes_from_file('./raw_packet/dns_response.raw')
        msg = DNSMessage.from_wire(dns_response_bytes)
        b = msg.encode_bytes()
        msg2 = DNSMessage.from_wire(b)
        assert msg == msg2
```

DNS Server

Basic design

The DNS server implements a LocalDNS server that performs iterative queries and handles special cases where the Answer section only contains CNAME records, queries with no Answer, and the

Additional section.

The main design and implementation of this project is the iterative_query method, which takes a DNS request and the requesting IP address. Internally, it performs iterative queries and ensures that it ultimately returns a DNS response with an A type RR corresponding to the requested domain name in the Answer section, thus achieving the functionality of iterative querying.

Handling the case where the Answer section contains only CNAME records

If the response code only contains CNAME type RRs, it indicates the need to further query the IP address of the alias domain. The code implementation for this functionality is as follows.

```python
if len(resp_message.answer) > 0:  # got answer
    cname = None
    for rr in resp_message.answer:
        if rr.type == TYPE_A:
            return response_data
        elif rr.type == TYPE_CNAME:
            cname = rr.rdata
    # only cname
    if cname:  # send another request to cname field and get question
        cname_data = self.iterative_query(self.construct_query(cname).encode_bytes(), self.root_dns_address)
        cname_msg = DNSMessage.from_wire(cname_data)
        add_an_len = len(cname_msg.answer)
        resp_message.header.ancount += add_an_len
        resp_message.answer.extend(cname_msg.answer)
        return resp_message.encode_bytes()
    else:
        raise 'error! no cname for final select'
```

In this code, it checks if there is an A type RR in the response. If found, it terminates and returns the response byte stream directly. If only CNAME type RRs are found, it calls the iterative_query method again to query the IP address of the CNAME domain from the root node. Since iterative_query ensures to return a byte stream containing A type query results, the results can be added directly to the Answer section of the previous DNS packet, then encode_bytes is called to re-encode and return it.

Since the functionality to encode a DNS object into a byte stream has already been implemented earlier, it is relatively easy to implement the construct_query method to construct a query request for a specific

domain name and encode it. By parsing->adding to the Answer section->re-encoding, the query results can be added. Therefore, this project implements the process of handling this special case.

Testing Result

1. passing test.py

```
if __name__ == '__main__':
    unittest.main()
```

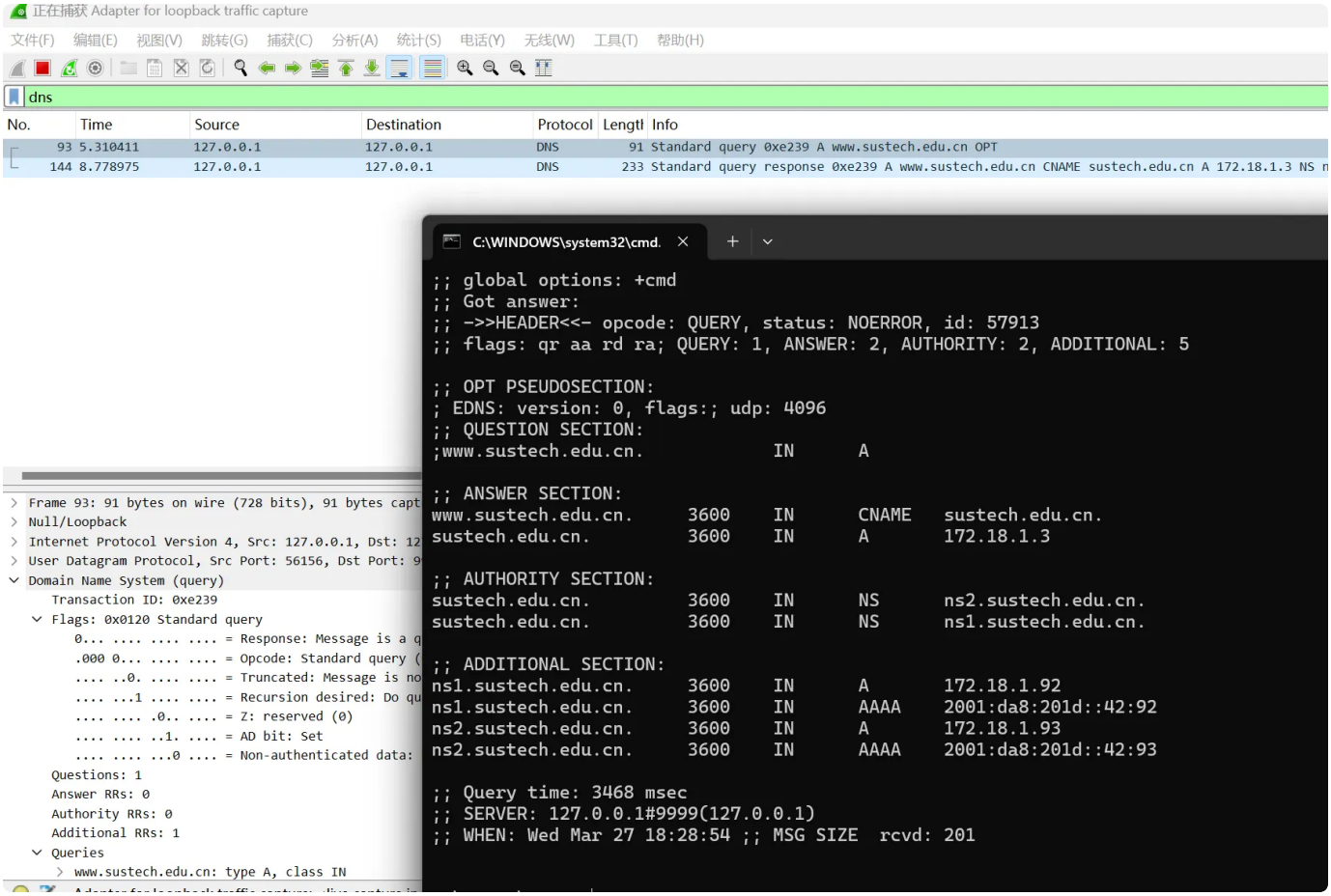
TestDNSResolver › test_header()

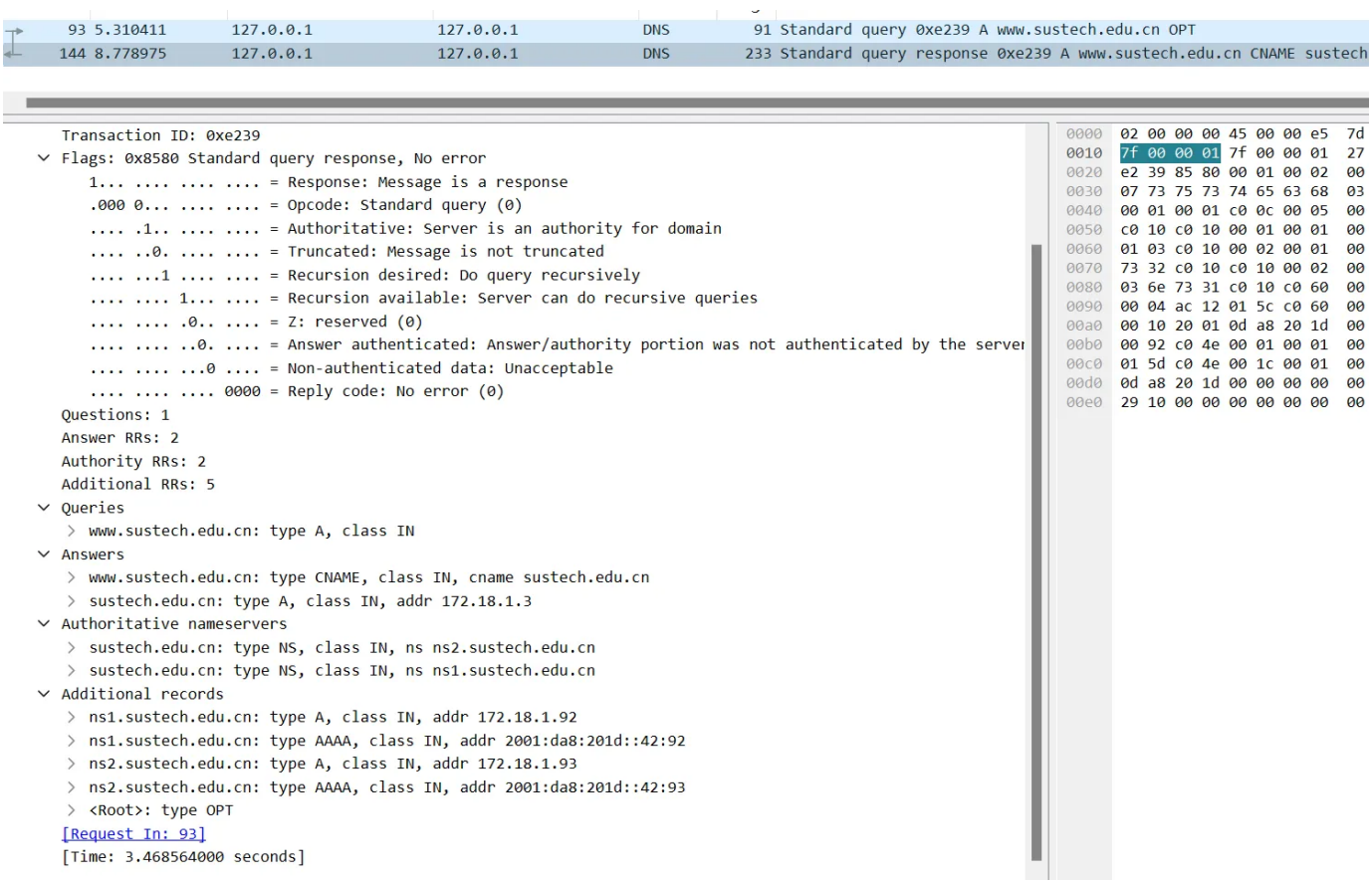
Tests passed: 3 of 3 tests – 1 ms

```
E:\ProgramFiles4\Python\Python380\python.exe "E:/ProgramFiles4/PyCharm
2023.3.4/plugins/python/helpers/pycharm/_jb_unittest_runner.py" --path
D:\ComputerScience\Projects\Computer_network_PA1\Project\test.py
Testing started at 18:25 ...
Launching unittests with arguments python -m unittest
 D:\ComputerScience\Projects\Computer_network_PA1\Project\test.py in
 D:\ComputerScience\Projects\Computer_network_PA1\Project

**header test passed**
**question test passed**
**whole msg test passed**


Ran 3 tests in 0.003s

OK

Process finished with exit code 0
```

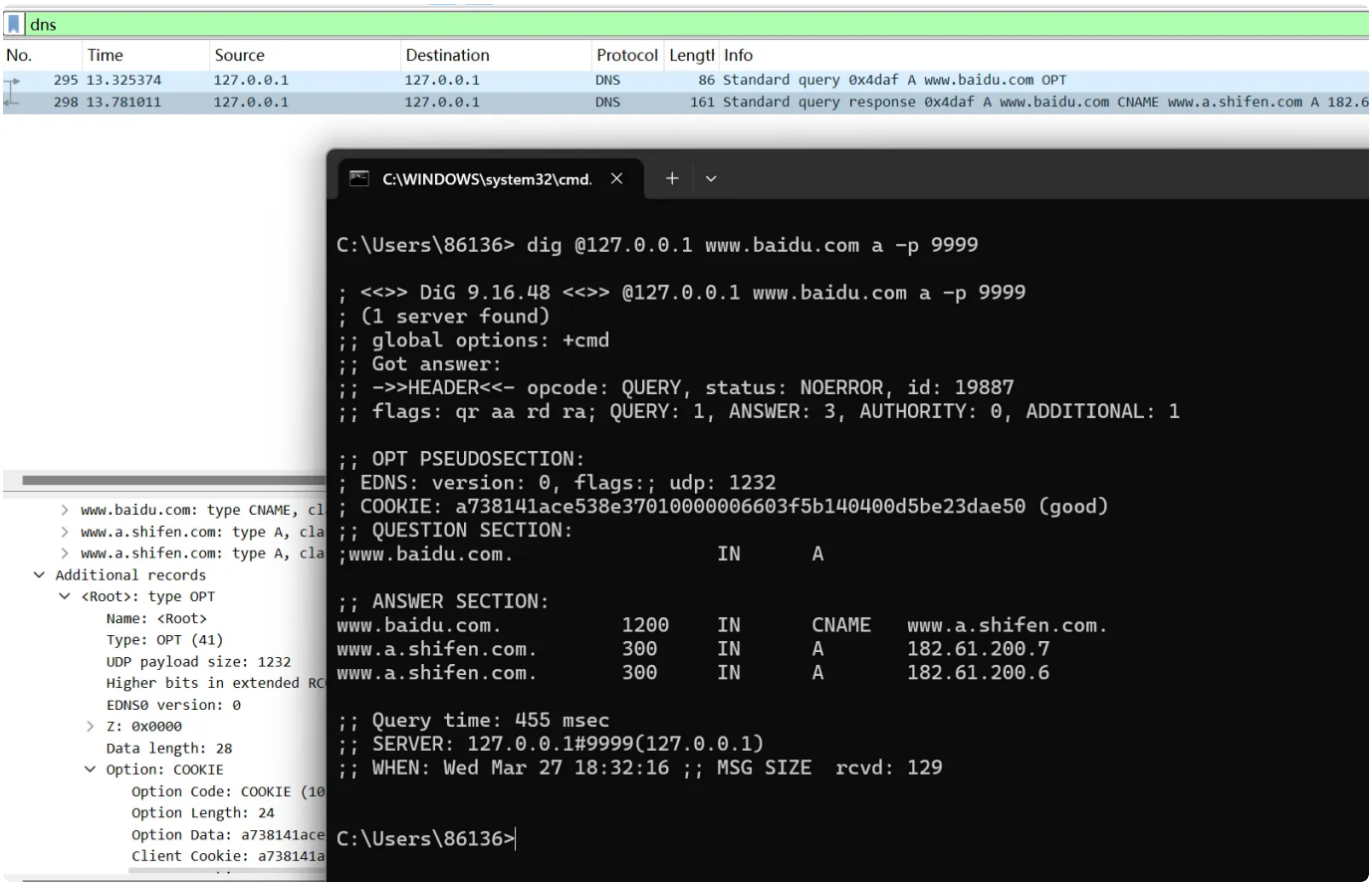
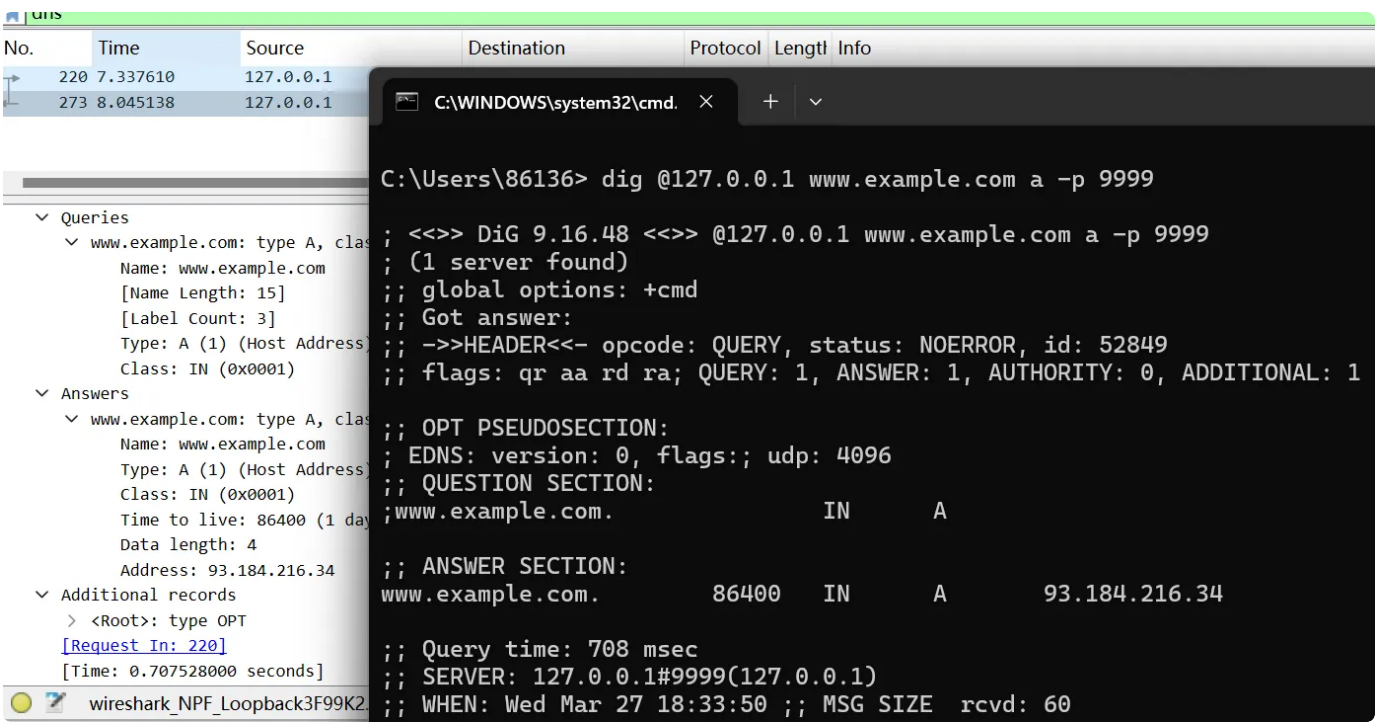
2. The packet capture results from Wireshark

Result of www.sustech.edu.cn





Result of www.baidu.com



Result of www.example.com