DNS Server Implementation

ZID z5411219

Codebase

Language: Python 3.12

Platform: MacOS, CSE environment

Main files:

- `server.py` (include all main logic for the server, including initialisation, multithreading, caching, resolving queries and logging)
- `client.py` (include all main logic for the client, including initialisation, multithreading, sending queries and displaying output)
- `classes.py` (include all constants, dataclasses and helper functions used including DNS header, payload, record and response. includes dataclass functions to parse to and from bytes)

Program design

Server

In this assignment, the server acts like a proxy node to help us resolve DNS queries from client. A typical server needs to understand the query from client, perform a DNS query, receive a response from local domain name service (in our case we utilise a master file and a cache), parse and understand the response, and send the answer/response to the client.

The server first initialises a UDP socket by defining the socket for the server side and binding it to the loopback address `127.0.0.1` . The server loads the master file into its cache, such that we query the cache for future queries. It then runs and listens for requests from client.

Caching requirements

In an actual caching scenario, every time a server receives a response from DNS queries, it temporarily caches the entry. First it lookups the cache file and if the query is found, the server directly responds the client. Or else, it perform a DNS query to a local domain name service.

In a simplified version of the cache, our server maintains a simple in-memory cache implemented using a dictionary. Unlike real caches, the TTL value is omitted to simplify caching for this assignment. As opposed to using a list, or continually reading from the master file, a dictionary provides much **faster lookups**. This is because lookup in a dictionary has an average time complexity of O(1) regardless of the size of the dataset. We can also store multiple records of the same type (e.g. multiple A types) for one name, which is good for iterating over them.

The cache is in a **nested dictionary** format, the first key being the query name and the second key being the type. This allows us to search for all records matching query name, or specifically search for a name and type.

```
class DNSCache:
    def __init__(self):
        self.cache = {}

    def get_cache(self):
        return self.cache

    def add_record(self, qname: str, qtype: str, record: str):
        qname = qname.lower()
        if qname not in self.cache:
            self.cache[qname] = {}
        if qtype not in self.cache[qname]:
            self.cache[qname][qtype] = []
        self.cache[qname][qtype].append(record)

    def get_records(self, qname: str, qtype: str) -> list:
```

```
        qname = qname.lower()
        return self.cache.get(qname, {}).get(qtype, [])
```

Multithreading

There exists a main thread to listen for new queries and when a query is received the server spawns a new thread using Python's `threading` module, which will sleep for the delay duration, process the query, send a response, and then terminate when the target function (`handle_query`) finishes executing. This allows the server to multiplex queries from multiple clients.

There are no race conditions as the threads do not write to the cache, however that could be something implemented in the future.

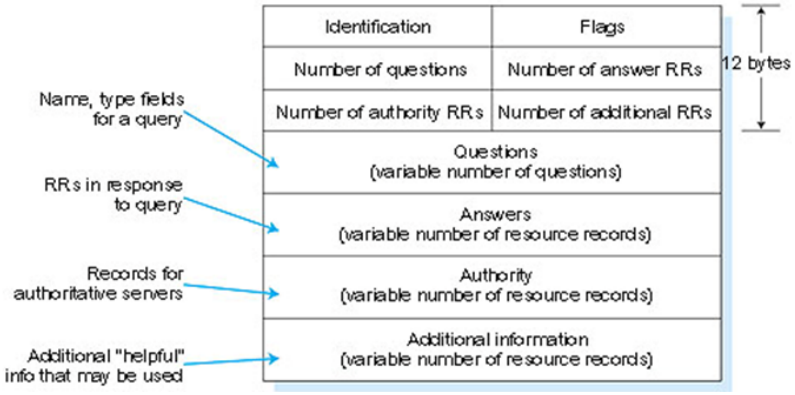
```
incoming_message, client_address = self.server_socket.recvfrom(BUFFERSIZE)
thread = threading.Thread(target=self.handle_query, args=(incoming_message, client_address))
thread.start()
```

```
delay = random.randint(3, 9)
print(f"{received_time.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]} rcv {client_address[1]:<5}: {header.qid:<4} {

time.sleep(delay)

response = self.process_query(header.qid, question)
self.server_socket.sendto(response or b"", client_address)

sent_time = datetime.datetime.now()
print(f"{sent_time.strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]} snd {client_address[1]:<5}: {header.qid:<4} {ques
```

DNS Query and Response

We parse incoming queries using a shared dataclass between the server and the client. A **dataclass** was chosen in favour of named tuples, fstrings, or dictionaries. This is because dataclasses are easily mutable, allows for default values, and you can use names to refer and retrieve different fields, making the code more readable and maintainable overall.

I briefly considered using a UDP header format but later research made me realise I should follow the format of a DNS message.




```
FLAG_QUERY = 0
FLAG_RESPONSE = 1

@dataclass
class DNSHeader:
    qid: int  # 16-bit unsigned integer as an identifier for the query
    flags: int  # Flags to represent various settings (e.g., query/response)
    num_questions: int = 0
    num_answers: int = 0
```

```
    num_authorities: int = 0
    num_additionals: int = 0

@dataclass
class DNSQuestion:
    qname: str  # target domain name of query
    qtype: int  # type of the query

@dataclass
class DNSRecord:
    name: str  # domain name
    type_: int  # type of the resource record
    data: str  # type-dependent data which describes the resource
```

Header format

The query and response share the same 12-byte DNS header. The fields were chosen to closely mirror the DNS header structure outlined in the RFC and each field has a specific purpose. We store the `qid`, a 16-bit unsigned integer, randomly generated by the client to exist as an identifier for the query. The `flags` field is defined as either 0 or 1 to identify whether it's a query or response. The following count fields `num_questions`, `num_answers`, `num_authorities`, `num_additionals` are used to parse DNS responses, so we know how many RRs there are for each section of the response.

Encoding and decoding

Using dataclasses have the additional benefit of **modularity** and **encapsulation**, as I include functions to convert to and from bytes under each dataclass, so one can easily import and call these functions from either `client.py` or `server.py` . These functions convert Python objects into a byte string using the `struct` module. Occasionally, `to_bytes` is used. Firstly, converting to byte strings matches DNS protocol conventions, and secondly, binary data is much more **size-efficient**, especially when a limited buffer exists.

These functions **abstract** most of the logic of encapsulation and decapsulation from the client and server, allowing the server to focus on resolving queries.

```
@dataclass
class DNSHeader:
        # { fields here }
    def to_bytes(self) -> bytes:
        fields = dataclasses.astuple(self)
        return struct.pack("!HHHHHH", *fields)

    @staticmethod
    def parse_header(reader):
        # print('reader', reader)
        header_data = reader.read(12)
        if len(header_data) != 12:
            raise ValueError(f"Header data is not 12 bytes: {len(header_data)} bytes received. Data: {heade
        items = struct.unpack("!HHHHHH", header_data)
        return DNSHeader(*items)
```

Resolving queries

The steps are as follows: the server looks up its cache with the question name and type. In a real-world scenario, this speeds up response time because the server can avoid querying external DNS servers, which is a much longer process than looking up an in-memory cache. If no match is found and the type is not CNAME, it checks for CNAME records and follows the CNAME chain until a final A record is found or no further CNAMEs are available. This is because CNAME records map an alias to its canonical name, and the client needs to find the ultimate final destination IP address. If no answer is found for the query type, then the server searches for NS records matching the query name in the authoritative section and subsequently the additional section with corresponding A records. With the A records, the client can directly lookup the authoritative name servers without needing additional lookups.

The response is encoded and returned to the client.

```
@dataclass
class DNSResponse:
    header: DNSHeader
    question: List[DNSQuestion]
```

```
    answer: List[DNSRecord]
    authority: List[DNSRecord]
    additional: List[DNSRecord]
```

Client

First, the client creates a UDP socket using `socket.AF_INET` and `socket.SOCK_DGRAM`. The socket is not explicitly bound to any port, so the operating system assigns an **ephemeral port** for this socket.

```
self.client_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
```

Given a server port, query name, query type and timeout, a client query simulates a DNS query. A query dataclass is constructed with the format of a DNSHeader plus a DNSQuestion serialised into bytes so we can transmit it to the server through UDP.

```
@dataclass
class DNSHeader:
    qid: int  # 16-bit unsigned integer as an identifier for the query
    flags: int  # Flags to represent various settings (e.g., query/response)
    num_questions: int = 0
    num_answers: int = 0
    num_authorities: int = 0
    num_additionals: int = 0

@dataclass
class DNSQuestion:
    qname: str  # target domain name of query
    qtype: int  # type of the query
```

The timeout is the duration (in seconds) the client should wait for a response before considering it a failure. This is to prevent the client infinitely waiting for a response. This is done by setting a timeout for the socket and waiting for a response.

```
self.client_socket.settimeout(self.timeout)
# { ... }
except socket.timeout:
    print("Request timed out")
    self._is_active = False  # close the sub-thread
```

Multithreading

Each query is processed in its own thread, preventing blocking and ensuring the server remains responsive to new requests.

```
#  on initialisation, start the listening sub-thread first
self._is_active = True  # for the multi-threading
self.response_received_event = threading.Event()
self.listen_thread = Thread(target=self.listen)
self.listen_thread.start()
```

```
def listen(self):
    """(Multithread is used)listen the response from receiver"""
    while self._is_active:
        try:
            self.client_socket.settimeout(self.timeout)
            incoming_message, _ = self.client_socket.recvfrom(BUFFERSIZE)
            self.handle_response(incoming_message)
            self.response_received_event.set()
            self._is_active = False  # Stop listening after receiving the response
        except socket.timeout:
            print("Request timed out")
            self._is_active = False
        except OSError as e:
```

```
            if not self._is_active:
                # Socket was closed as expected
                break
            else:
                raise e
```

I increased the timeout of the listener a little bit to allow the capture of timeout events.

```
def run(self):
     self.create_and_send_query()
     self.response_received_event.wait(self.timeout+2) # some more time to listen for response/timeouts
     self._is_active = False  # close the sub-thread
     self.client_socket.close()
     print("Socket closed.")
     self.listen_thread.join()
```

I acknowledge multithreading is a bit overkill for this simple client-server model. However, this allows the listening thread to continuously monitor for incoming messages, while the main thread handles user interaction and query creation. It's also closer to a real-world implementation of DNS, which might need concurrency within the client itself.

Known Limitations

Several functions make use of explicit error checks or try-catch blocks, but the client and server do not robustly handle **error cases** mainly due to the nature of the assignment. More descriptive and robust error handling is good practice for any coding task.

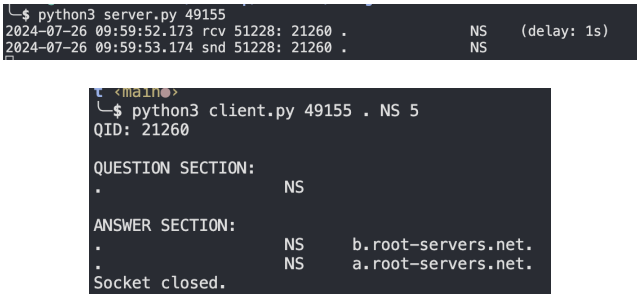
As I only decided to move my encoding and decoding functions to a dedicated file halfway through the assignment, I tried my best to refactor the code to make better use of OOP principles. However I do acknowledge the code could be cleaner, and the encoding and decoding functions could be more consistent, as some use `struct.pack` / `unpack` , some use `to_bytes` , some use a `BytesIO` buffer etc.

Code References

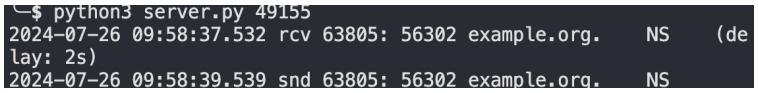
- I modified the template demo code from Rui Li (Tutor for COMP3331/9331) to start the assignment:
  - https://github.com/lrlrlrlr/COMP3331_9331_23T1_Labs/tree/main
- I referenced concepts and data structures from Julia Evans's "Implement DNS" guide:
  - https://implement-dns.wizardzines.com/book/part 1
- I referenced the RFC 1035 Internet Standard for several constant values
  - https://datatracker.ietf.org/doc/html/rfc1035#section-3.2.2

Appendix: Sample output

G.4. . NS



G.9. example.org. NS



```
└─$ python3 client.py 49155 example.org. NS 5
QID: 56302

QUESTION SECTION:
example.org.        NS

AUTHORITY SECTION:
.                   NS      b.root-servers.net.
.                   NS      a.root-servers.net.

ADDITIONAL SECTION:
a.root-servers.net. A       198.41.0.4
Socket closed.
```

G.10. www.metalhead.com. A

```
└─$ python3 server.py 49155
2024-07-26 09:56:08.378 rcv 60191: 4390 www.metalhead.com. A     (delay: 0s)
2024-07-26 09:56:08.379 snd 60191: 4390 www.metalhead.com. A
```

```
└─$ python3 client.py 49155 www.metalhead.com A 5
QID: 4390

QUESTION SECTION:
www.metalhead.com.  A

ANSWER SECTION:
www.metalhead.com.  CNAME  metalhead.com.

AUTHORITY SECTION:
com.                NS     d.gtld-servers.net.

ADDITIONAL SECTION:
d.gtld-servers.net. A      192.31.80.30
```