Design and Implementation of

A DNS RELAY

Contents

1. Overview

1-1 Brief Introduction

DNS, the abbreviation of “Domain Name System”, is a system that composed of a resolver and a domain name server. The domain name system assigns domain name addresses and IP addresses to hosts on the Internet. If the user uses a domain name address, the system will automatically convert the domain name address to an IP address. The domain name service is an Internet tool that runs the domain name system. The server that performs the domain name service is called the DNS server, and the DNS server answers the query of the domain name service.

The domain name server converts domain names and corresponding IP addresses. A table of domain names and IP addresses corresponding to them is stored in DNS to resolve the domain names of messages. It can also be called a method of managing names, which is to divide the names of subsystems into different groups. Each layer in the system is called a domain, and each domain is separated by a dot. The so-called domain name server is actually a host with a domain name system. It is a hierarchical structure database that can realize name resolution. In fact, every domain name must have at least two DNS servers, so that if one of the DNS servers has a problem, the other can also return data about the domain name. There can also be more than two DNS servers, but the DNS records on all these DNS servers should be the same.

1-2 Main Targets

During the whole lesson and practices, we mainly learnt acknowledgement related to DNS, Wireshark and practiced the socket programming.

DNS:

- DNS concepts and functions
- DNS hierarchy
- DNS message format
- DNS transfer
- DNS server

Wireshark:

- Able to configure Wireshark
- Able to achieve communication between server and client
- Able to capture data packets transmitted on the network

- Able to display data packet fields and their meaning
- Able to use it to diagnose network failures

Socket Programming:

- Receiving DNS queries from DNS client (Resolver) and forward them to a given DNS server.
- Receiving DNS responses from DNS server and forward them to the Resolver.
- Despite these, fulfill 3 cases below:
  1）for IP address *0.0.0.0*, sending back “no such name” (reply code =0011).
  2）for domain name included, send back corresponding IP address.
  3）for domain name not included, forward query to local DNS server.

2. REQUIREMENTS ANALYSIS

2- 1 User Interface Requirements

Do not have any user interface for user. Users are allowed to achieve precision searching in the command line window.

- Open the CMD tool
- Input the query involved the wanting domain name:
  `nslookup <domain name> <localhost IP>`
- The checking result will then be presented with detailed information
- Javadoc document viewing

2- 2 Environment Requirements

Operating System: Windows 10 Enterprise, macOS 10.15
Programming Language: Java SE 11
Supporting Software: Eclipse IDE, IntelliJ IDEA, Wireshark
Network Condition: BUPT-portal Wi-Fi well-connected

2-3 Functional Requirements

According to the **fundamental requirements** that build the basic working program, simple functions should be built to meet all point above, which is at the same time illustrated as a basic process:

1) Open socket and use port 53 to start the DNS service.
2) Read DNS server IP address and path of local database.
3) Print the detailed information illustrated in the window, including the DNS server IP, the actual local UDP port, etc.
4) Close the socket connection when throw the exceptions when error occur.
5) Close the socket connection when completed.
6) Keep listening port 53 until a packet is received and then repeat listening.

According to the **advanced needs** mentioned in the sample power point, the expected project product should satisfy all testing rules below:

1) For IP address 0.0.0.0, sending back “no such name” (reply code =0011).
2) For domain name included in the database, sending back corresponding IP address.
3) For domain name not included in the database, forward query to local DNS

server.

2-4 Detailed Requirements

2-4-1 Packet Parser

The packet parser is mainly responsible for the packets. Generally, it is used to parsing the attributes contained in the received packet. Furthermore, they could also:

1) Identify whether it is handling a query packet or a response packet.
2) Select the actual information concealed in the packet, such as the domain name.
3) Judge whether the domain name has a reflection in local DB. If not, more things should be done instead of simply sent it back.

2-4-2 Local DNS Server

This part is mainly to prevent unnecessary works, which is mostly focus on the domain name's analysis. To better support, marking variables are used and certain values are checked in time. As a consequence, what to send back? Whether to send back? What to do next?... are questions answered in this section.

1) When the domain name is found in the database, however the reflecting IP is 0.0.0.0. We modify the Flag in the header, Flag=0x8183(as the reply code=0011, just an addition to 0x8180 of the initial value, from my perception). We modify the Matched variable, Match=0. Then we have to return message such as "no such name" for explanation.
2) When the domain name is found in the database, and the IP address does not equal 0.0.0.0. We remain the Flag unchanged, Flag=0x8180. We modify the Matched variable, Match=0. In addition, part of response message should also be included.
3) Response the resolver an empty answer when receiving an IPV6 request packet.

2-4-3 DNS Relay

To extract the domain name of requested that do not contained in the local database, we have to forward them to the relay, which has the ability to obtain the correct response for the local server.

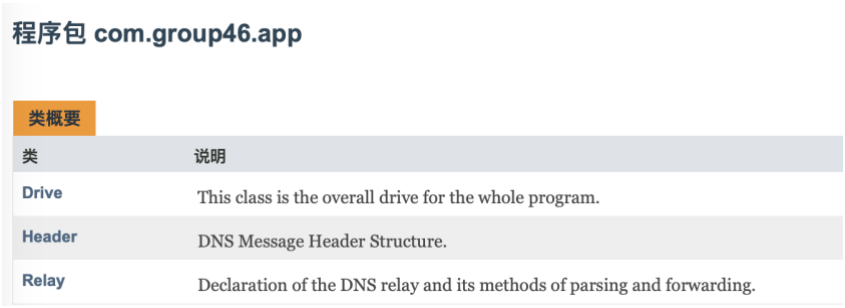
1) By setting the DNS server address, we then send packet to the server.
2) The server will send back its searching results as response message to the

localhost so that can be shown on the CMD tool.

3. PRELIMINARY DESIGN

3-1 Decomposition of functional modules

In this project, we have decomposed the whole program into three parts – Drive, Header and Relay app. Also, there is a useful “tools library” utils to assist main program.



程序包 com.group46.app

类概要

类	说明
Drive	This class is the overall drive for the whole program.
Header	DNS Message Header Structure.
Relay	Declaration of the DNS relay and its methods of parsing and forwarding.

Drive

The launching class for the program. Print out initialization information and create a thread-pool for the incoming query packet.

This class is the overall drive for the whole program. It contains a class static parameter, which will be initiated once the program is launched. This will be used as a communication media, which enables client to send query packet, and also enables server to send response packet back to client.

In terms of the main method for the Drive: It will firstly print welcome information on the console and initiate a DatagramPacket ready for the socket to receive incoming queries from client. It will then use Executors to create a single-thread's pool that will execute functionalities within Relay.

In this program we don't use any arguments.

构造器概要

构造器

构造器	说明
Drive()	

方法概要

所有方法 静态方法 具体方法

修饰符和类型	方法	说明
(专用程序包) static java.net.DatagramSocket	getRelayProgramSocket()	
static void	main(java.lang.String[] args)	Main method for the Drive.

从类继承的方法 java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

****

Header

The definition of DNS message, showing the DNS Message Header Structure.

It defined variables according to ID | flags (QR | QPCODE | AA | TC | RD | RA | Z | Rcode). We will make a clearer description next.

构造器概要

构造器

构造器	说明
Header()	

方法概要

所有方法 实例方法 具体方法

修饰符和类型	方法	说明
byte[]	getQuestionCount()	
void	setAdditionalCount(byte[] Additional_count)	
void	setAnswerCount(byte[] Answer_count)	
void	setAuthorityCount(byte[] Authority_count)	
void	setFlags(byte[] Flags)	
void	setQuestionCount(byte[] Question_count)	
void	setTransactionId(byte[] Transaction_id)	

****

Relay

Declaration of the DNS relay and its methods of parsing and forwarding. The main functionality of the relay is declared and implemented there.

构造器概要

构造器

构造器	说明
`Relay(java.net.DatagramPacket packet)`	After receiving packet from client, parse and load info of the packet into class variables for further usage.

方法概要

所有方法 | 实例方法 | 具体方法

修饰符和类型	方法	说明
`private int`	`readDatabase()`	Uses `LOCAL_DNS_DB_PATH` to load dns relay text file (database) into the static variable `database`.
`void`	`run()`	Overridden method for this runnable class.
`private java.util.ArrayList<java.lang.Object>`	`setHeader()`	Initiates a DNS `Header` object.
`private void`	`writeBuff(byte[] buff, int off)`	Uses `System.arraycopy(Object, int, Object, int, int)` to load data from byte array and copy them into the specific area in class variable `data`.

In terms of the RUN method: overridden method for the thread.

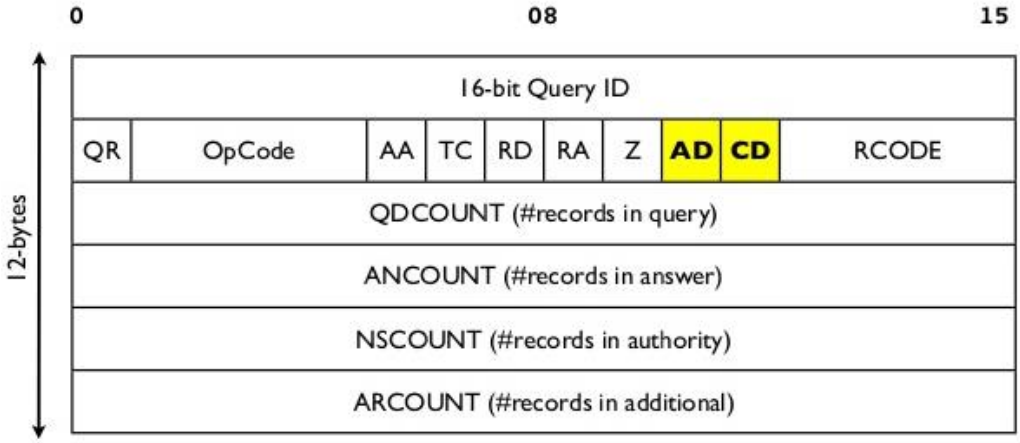
- Declares every functionalities for the dns relay, including loading domain name-ip database, setting header (by invoking private method setHeader()).
- Getting user-inputted domain name (by invoking static method from parseDomainName(byte[], int));
- Getting dns packet type; Getting DNS packet classification;
- Determining whether the user-inputted domain is hit in the local database (so that branching into different cases);
- Printing client request basic info;Parsing request;
- Designating response packet for the 3 cases;
- Finally sending back response packet to client via the relayProgramSocket that declared earlier in the Drive.class.

3-2 Message Structure Analysis

3-2-1 header

To be more specific, we defined all variables in Header.class according to the structure shown below.

In addition, the reason why we need to dig out the header separately is not only to modify the FLAG value when packing the response package, but also to set something like a mark, telling the programmer “we just finished reading all header’s part, then we come to the 4 section next!”



[Here attach the actual functions of each elements in the header below]

ID: The 16-bit flag set by the requesting client. When the server gives a response, it will return with the same flag field, so that the requesting client can distinguish between different request responses.

QR: Used to distinguish whether it is a request or a response.

OPCODE: 4 bits are used to set the type of query. When responding, it will bring the same value. Available values: 0 QUERY, 1 IQUERY, 2 STATUS, 3-15 reserved value, not used temporarily.

AA: Authoritative Answer-This bit is meaningful when responding, indicating that the server that gave the response is the authorized resolution server that queries the domain name.

TC: Truncation-Used to point out that the message is longer than the allowed length, causing it to be truncated.

RD: Recursion Desired-This bit is requested to be set, and the same value used in response is returned. If RD is set, it is recommended that the domain name server perform recursive resolution, and the support of recursive query is optional.

RA: Recursion Available-This bit is set or cancelled in the response to indicate whether the server supports recursive query.

Z: Reserved value, not used temporarily. Must be set to 0 in all request and response messages.

RCODE: Response code-These 4 bits are set in the response message, which means: 0 No error, 1 Format error, 2 Server failure-The request cannot be processed because of the server, 3 Name Error-parsed The domain name does not exist, 4 Not Implemented, 5 Refused, 6-15 are reserved values and are not used temporarily.

QDCOUNT: Indicates the number of problem records in the request segment of the message.

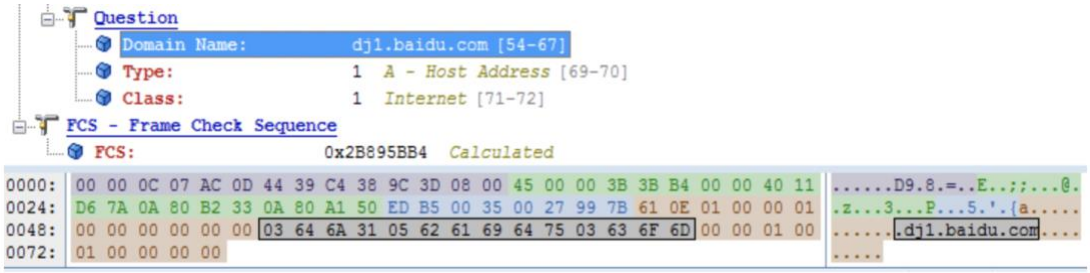
ANCOUNT: Indicates the number of answer records in the answer section of the message.

NSCOUNT: Indicates the number of authorization records in the authorization segment of the message.

ARCOUNT: Indicates the number of additional records in the additional segment of the message.

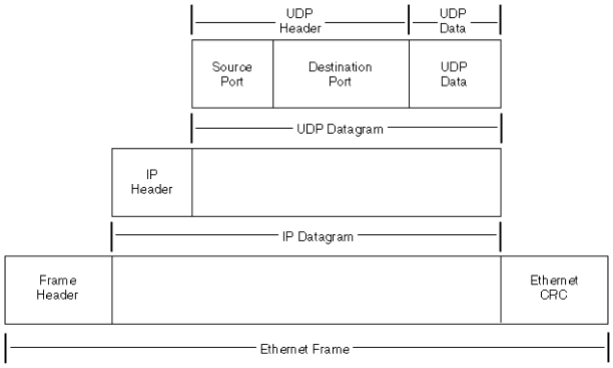
3-2-2 Question Section

The Question Section, which stores origin information from the received pack (including query domain name, query class, query type) remains unchanged in our program, unless the domain name cannot be found (we will forward it out as a special phenomenon).



For further discussion, we put the evidence it shown in the Wireshark to make it clearer. In the TCP/IP hierarchy structure, the upper-level messages are often contained in the complicated lower-level messages. As the sequence presents: MAC-IP-UDP/TCP.

1) We need to extract information about MAC in the Ethernet frame- its header at first. *[PURPLE part in the Wireshark screenshot]*
2) Then, we come to the IP part, selecting out the IP's header.
   [GREEN part in the Wireshark screenshot]
3) Next, it comes to the UDP protocol part, including the source port, destination port, packet length and the checksum.
   [BLUE part in the Wireshark screenshot]
4) Finally, we arrive at the DNS part.
   [RED part in the Wireshark screenshot]



All bytes in front of the GREY bytes are representing the header we mentioned above. As the graph suppose, there are 12 bytes in all, which fits our theorical foundation perfectly. Now, let's start to read:

03 64 6A 31 05 62 61 69 64 75 03 63 6F 6D

03 d j 1 05 b a i d u 03 c o m

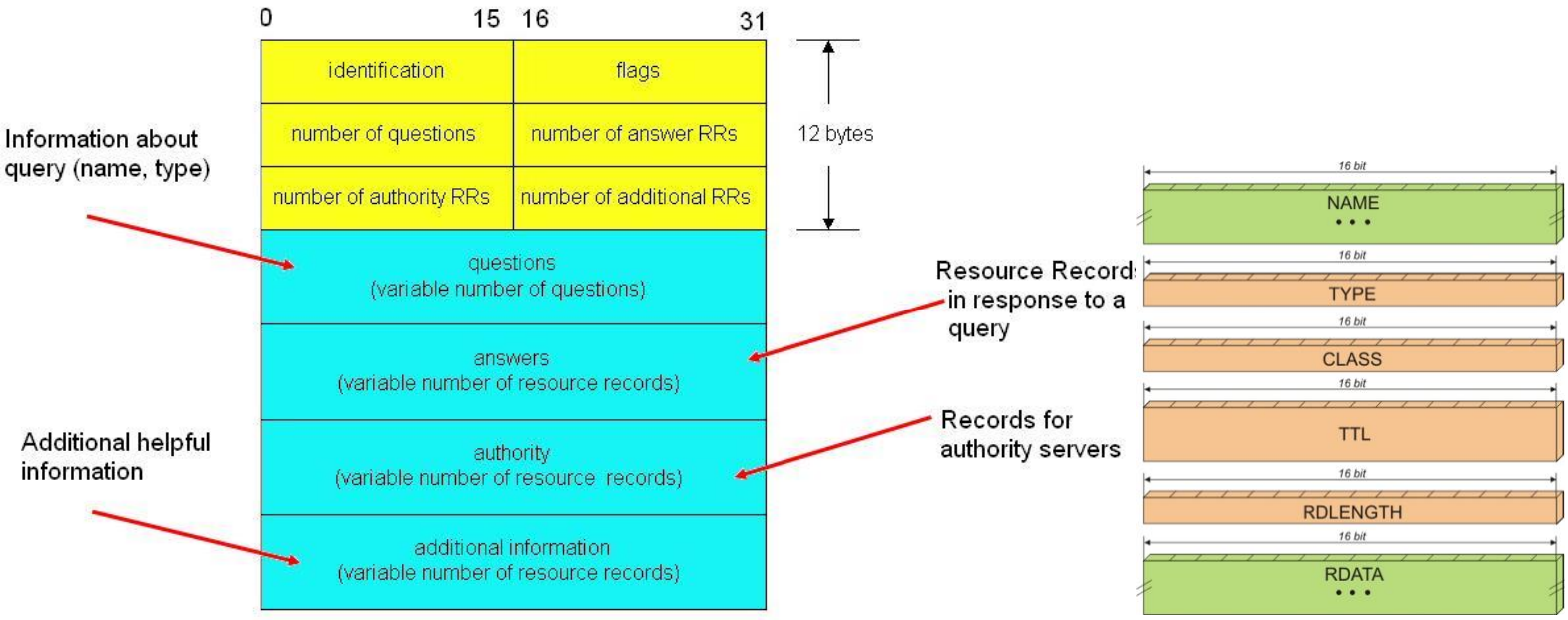
That is: jdi . baidu . com

This is the process we get the domain name.

3-2-3 Response Message

As the graph illustrates, the response packet is quite similar to the received one, especially for those in front of the Answer Section.

The yellow part on the top shows the header, which is the same as the one we mentioned in 2-4-1. The Question Section contains query information, such as domain name, class and type, which remains unchanged.



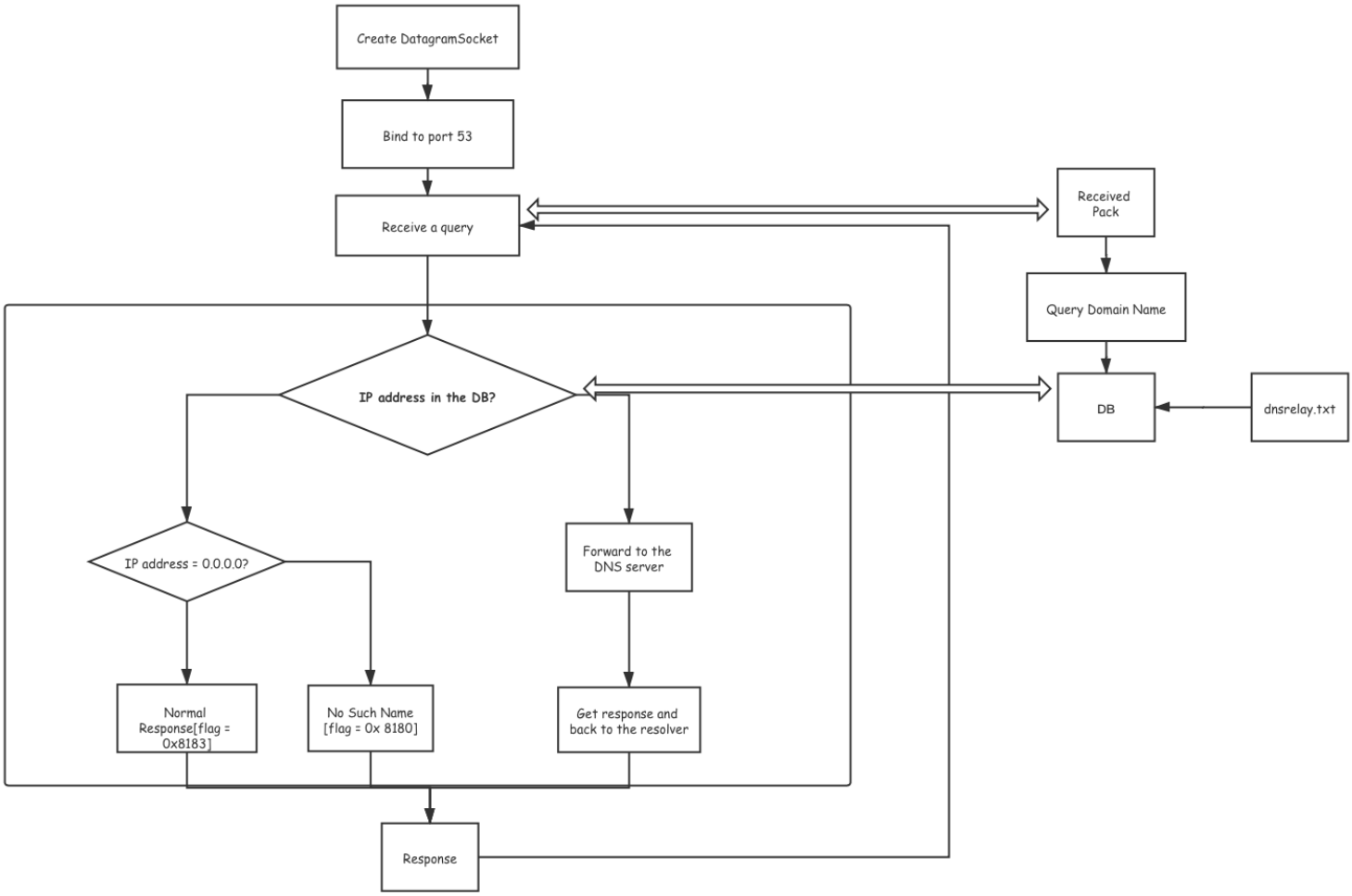
It is worth mentioning that the greatest change takes place in the following part. Looking at the structure next, we acknowledge that it still requires TTL, data length and response data provided to be completed.

[Related code is attached above.]

```
System.arraycopy(int2byte(3600 * 24, 0), 0, response, offset, 4);
offset += 4;
System.arraycopy(int2byte(4, 1), 0, response, offset, 2);
offset += 2;
System.arraycopy(Objects.requireNonNull(string2byte(IP)), 0, response, offset, Objects.requireNonNull(string2byte(IP)).length);
offset += Objects.requireNonNull(string2byte(IP)).length;
```

4 DETAILED DESIGN

4- 1 Overall Flow Chart



4-2 Declaration

To get start, we need to extract the specific part, header, which plays a significant role in the whole process, from the initial packet. As the header contains quite a numerous of variables, such as transaction ID, flags, question count, …, etc. To clarify this part, as well as split them apart from the main body (or easier to be confused), we declared a new class for that.

The whole class is consisted of 4 parts, just as most of official declarations do.

1) We made definitions for temporary variables, showing, giving 2 bytes size for each variable

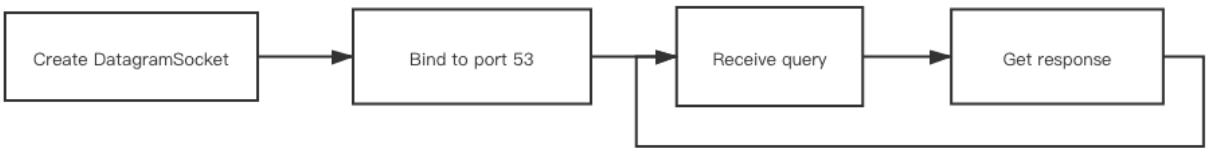
according to the official instructions.

2) The constructor is also needed even if we left it empty, as the initial one does not contain any passing value.
3) To makes the actual constructor works, we overwrote the constructor, called Header(…). In this way, we allowed the passing external values interact with this class. By defining unused variables with real values outside, one-by-one, more complicated functions are enabled as follow.
4) The most significant part is then followed- the function section to make it works. Considering the response message just need another header, we will get values from the received one for some comparation, and also set new values for constructing the new header.
   To be more specific, we should extract all variables' values from data collected from the received pack.
   - First of all, the prior 2 bytes are collected. We write them down at once on the new header's proper place, using THIS to represent the current state value. (This step shows the way to write the SET function) To write the GET function, we simply use RETURN.
   - Similarly, the following steps to get & set other variables are quite the same.
   - …

   Despite those above, it is also necessary to declare new variables such as packet, data, socket, database and so on for preparation in advanced.

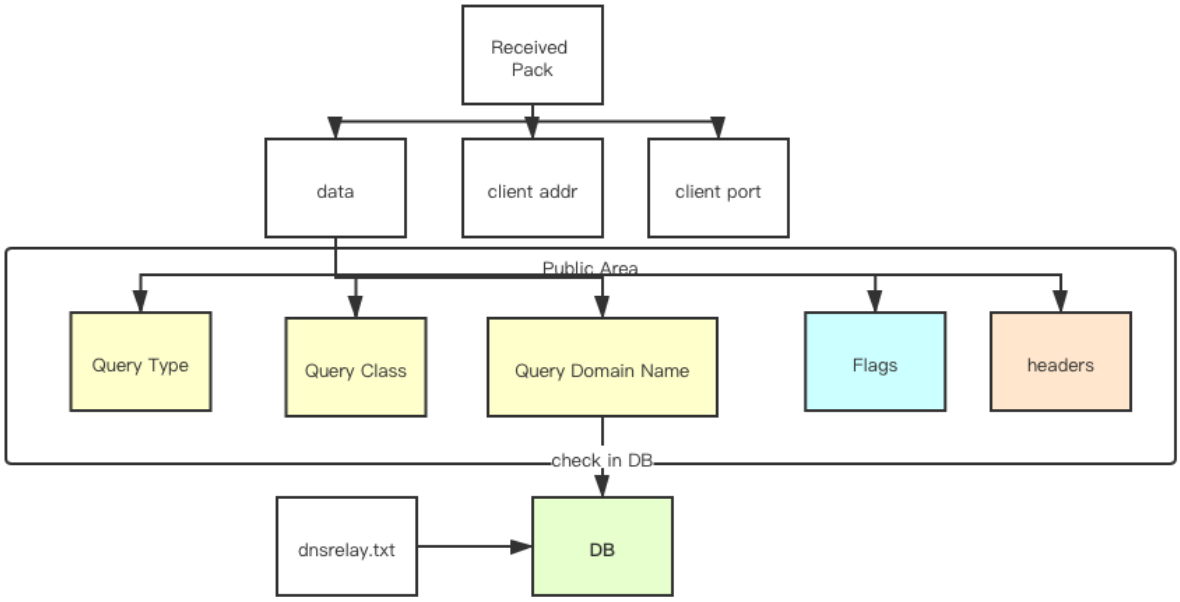
4-3 Initialization

1) Create the socket of DNS server type Datagram Socket and initialize.
2) Create the method get Socket to support the final pack sending process.
3) Create the UDP packet and bind to port 53, making it available. Besides, put this section into a try-catch procedure to guarantee the necessary exception when error occurs.
4) Creating multiple threads, in order to get prepared for frequent request when the previous response not yet generated. (We delete this part in the latest version code)
5) Put the major process into a WHILE loop – enabling the socket to get received packets, then execute them separately in distinct threads. That is to say, whenever it comes a query, we handle it in the thread. Generally, only one thread would be used at the same time, but there is a possibility that more than one threads are used if the previous one did not get response.
6) Create a count variable to present the working thread.



4-4 Preparations for Response

1) Do something with the dnsrelay.txt. Put all data into an array as a database. Do preprocess like splitting and storing.
2) Get necessary information from the received packet. This includes, client address, client port, data… In practical design, all those above could be extracted by using embedded packet's methods.
3) By using previous header class, get the header and its detailed information from the packet.
4) Analise the Question section – do some processing according to text characters and extract the query domain name, query type and query class as following.



4-5 Classifications of Domain Name

Judge the correct classification of domain name:

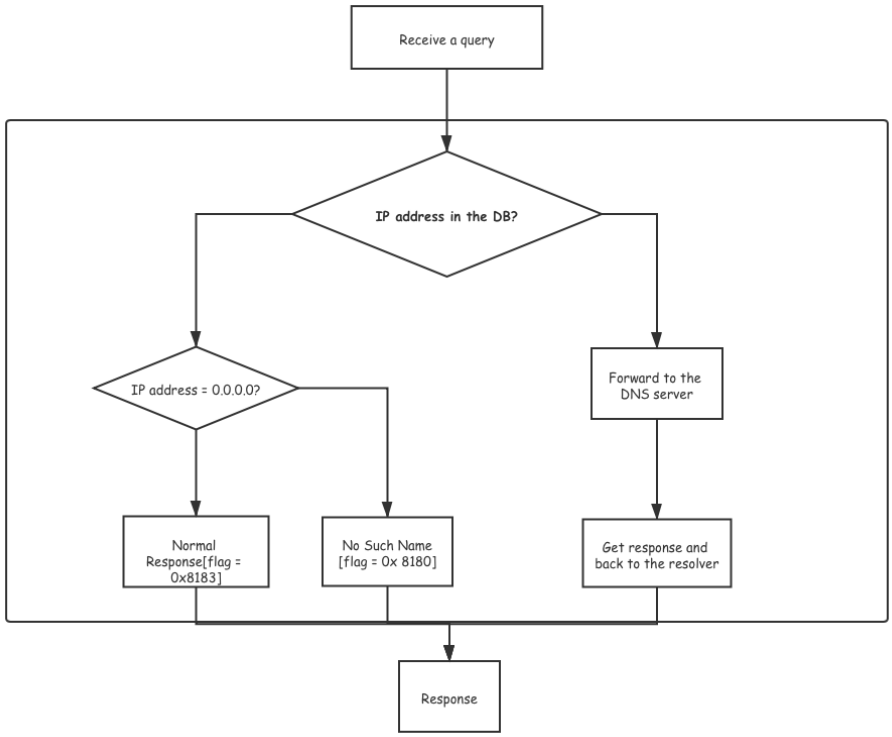
Whether the IP could be found in the local database?

- **IF yes,** match = 1, put IP according to the one in the database.
  Whether the IP equals to 0.0.0.0?
  - **IF yes,** shield the packet by change the flag into 8183
  - **IF no,** change the flag into 8180.

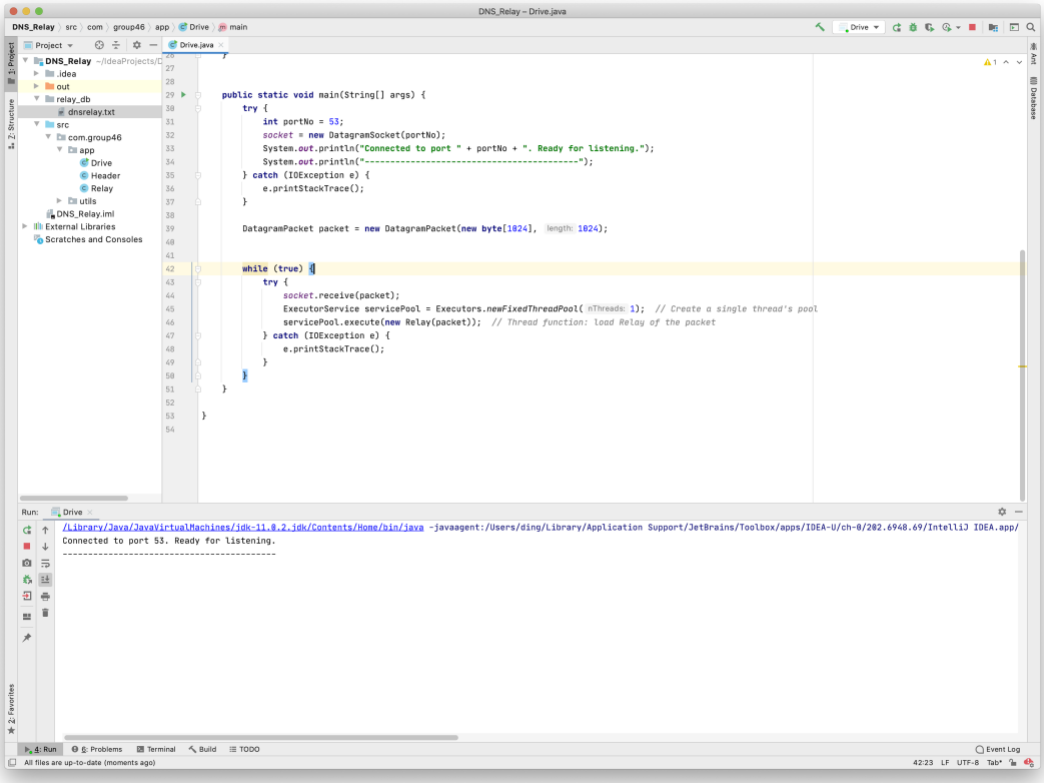
  Then make a UDP packet as a response.
  Send back to resolver by the address and port stored in Model 1
- **IF no,** just forward it to the DNS server.
  Create a new response socket type internet Socket. By sending the send pack, the response socket waits and receives the new packet. Then, we regard it as the response

packet. Finally, we send the response pack for return.

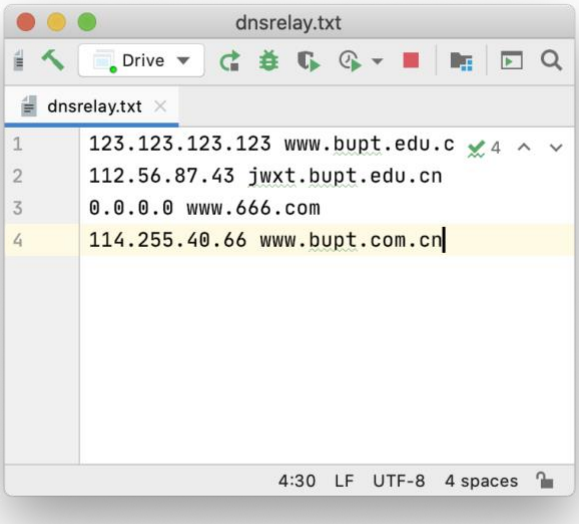
[Note: all processes above are included in a try-catch pair, in case there is any exception appears]



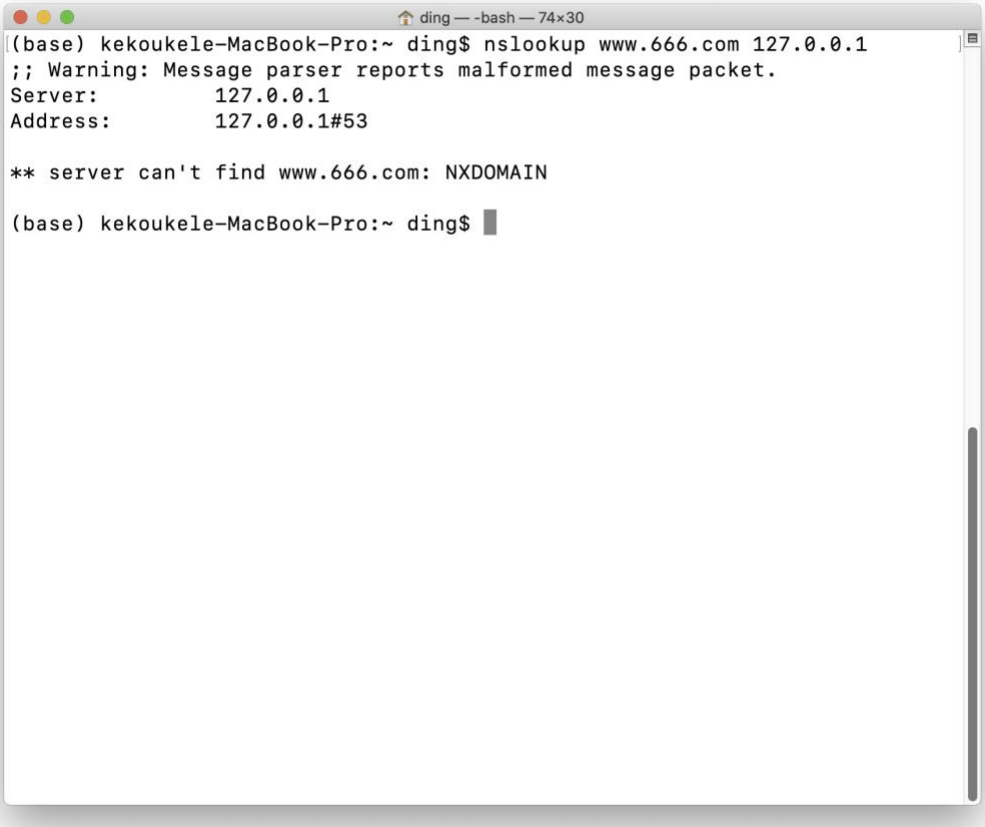
5. RESULTS



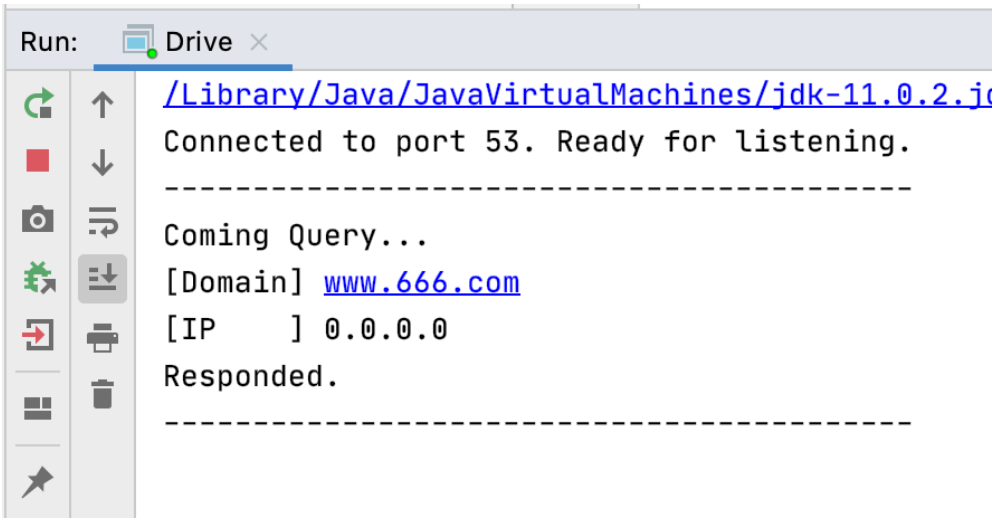
Start Drive and make it listen for coming queries

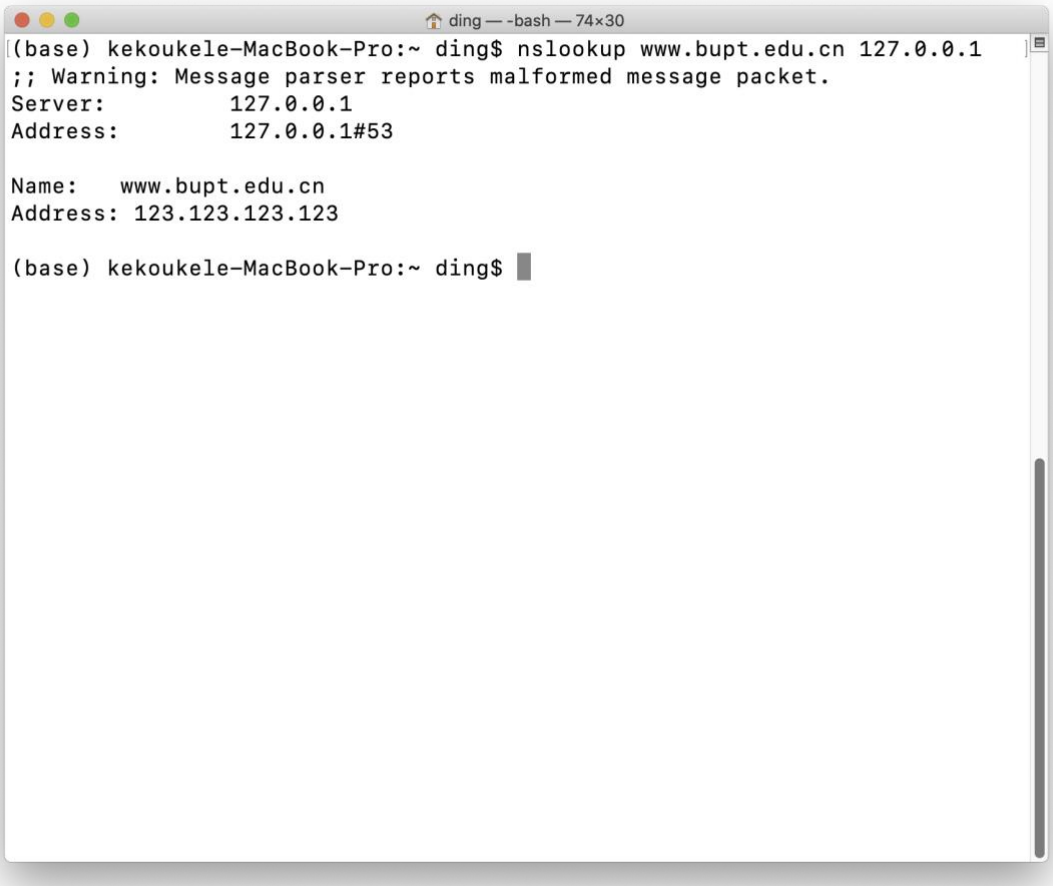


Local database

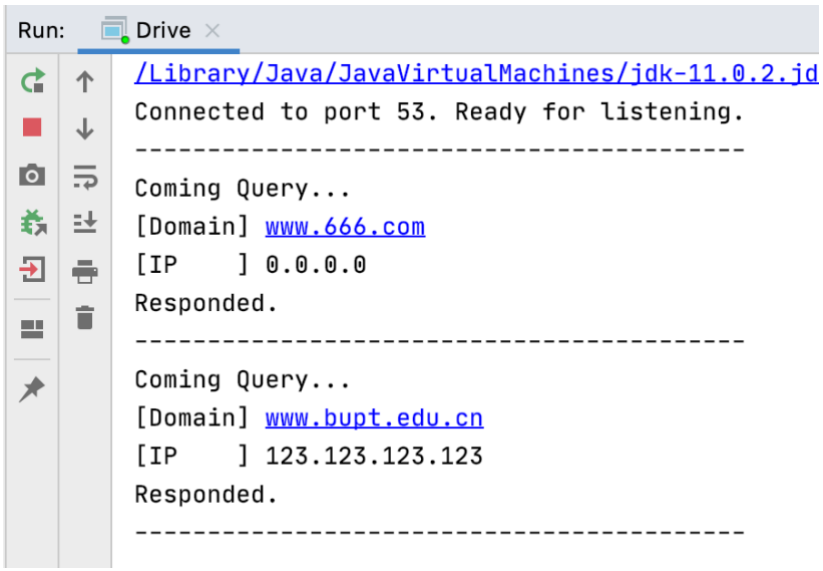

ding — -bash — 74×30
(base) kekoukele-MacBook-Pro:~ ding$ nslookup www.666.com 127.0.0.1
;; Warning: Message parser reports malformed message packet.
Server: 127.0.0.1
Address: 127.0.0.1#53
** server can't find www.666.com: NXDOMAIN
(base) kekoukele-MacBook-Pro:~ ding$

Case 1: 0.0.0.0

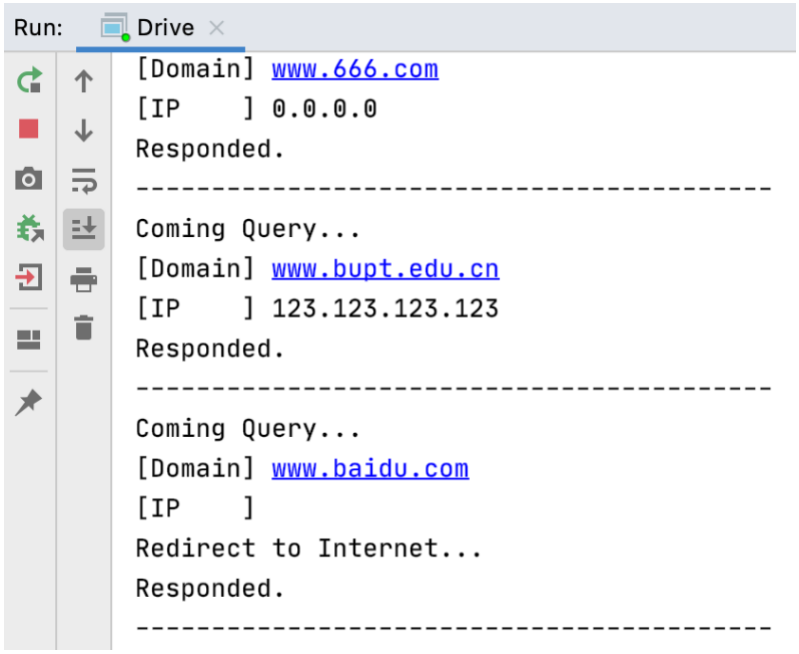

Run:
Drive
/Library/Java/JavaVirtualMachines/jdk-11.0.2.j
Connected to port 53. Ready for listening.
Coming Query...
[Domain] www.666.com
[IP] 0.0.0.0
Responded.



Case 2: local database



Case 3: not in local database



6. Problems Explanation [Additional]

First of all, we still would like to express our deeply apologize to you, for the unclear answer we presented during the face-to-face inspection on Sept 14th. We knew we made a mistake explaining the concept of thread, and here is our re-interpretation.

Now, let me make a brief description alternatively, asking for a second chance.
[**Note: all example below are based on the version we presented to you on Sept 14th.*
The latest version with many modifications are enclosed in our package.]

****

1-- "How many sockets did you use in this program?"

1) Basically, there is only one for basic use (as we insisted that day). The Datagram socket is the one that created to provide a communication platform for both client (which will send query request) and dns-relay (which will receive query, parse it, and send back corresponding responses). The socket was declared in port 53 as usually did.

```
DatagramSocket socket = new DatagramSocket(53);
```

2) However, there is also another condition:

The second socket is needed to forward a searching query - for those domain names that not included in the database, we have to forward the query to internet server. In this way, we wrote the code below:

```
DatagramSocket internetSocket = new DatagramSocket();
```

```
internetSocket.send(internetSendPacket);
```

Thus, there are two sockets in our design. But due to the role that the internetSocket plays, it is actually a dispensable socket in the whole program. That is to say, the socket becomes useful only when a certain domain name checked could not be found in the local database, while the previous one is always necessary. So, there is always "one" socket, but also maybe "two" sometimes.

****

2-- "Why you use multi-thread in this program? How it works?"

1) Why to use?

It will be reasonable for users to request new command when waiting. To be more specific, the multiple threads are meant to prevent users from waiting when they intend to send a new request just after the previous one. Although this situation not always happen, users can hardly feel friendly when being trapped until the response back. It will be even worse when other also find it difficult to send back the response.

2) How it works?

As you see, all the processes are contained in a while loop.

[the 1st line] Once you get a piece of "nslookup…" command, the socket receive a packet at the same time, whom is the one we then need to extract detailed information (including domain name) from.

[the 2nd line] Now we use an available thread to run the functions in response ( ) method, which is aimed at generating the response as a final purpose.

As long as we don't use up all the threads we set in advanced, everything goes correctly regardless of how slow each process is. Even under the worst situation when the previous packets are sent yet without reply, users could still send their new request as usual.

```
while (true) {
        try {
            socket.receive(packet);
            servicePool.execute(new Response(packet));
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```

****

3-- "What's the use of this FOR loop??" [Discussion and Modification]

Well, I truly feel sorry for our previous explanation, which was really ambiguous and confusing.

Actually, the FOR loop is only used to help calculating. That is to say, the existence of FOR loop will not influent its actual function.

```
while (true) {
    for (int k = 0; k < 10; k++) {
        System.out.println(k);
        try {
            socket.receive(packet);
            servicePool.execute(new Response(packet));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
while (true) {
        try {
            socket.receive(packet);
            servicePool.execute(new Response(packet));
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```

（the version during inspection with FOR VS the version without FOR now）

We can find the only difference between them is the output of **k**, but nothing more:

- **When there is only a WHILE loop**, the code provides services to receive packet and execute Response method in new thread. Mention that, each time we occupy a created thread, the WHILE loop comes to an end.

- **When we use the FOR loop in the WHILE**, the code also provides services to receive packet and execute Response method in new thread as it does in the previous one. But in this way, the execution part ends in the FOR loop instead- each time it ends, the k value plus 1. In addition, the maximum number “10” is just warning us not to get out of bound in display (but nothing more) as we just created 10 thread in all. Then, we could see the output just like “thread 0”, “thread 1”..etc.
  It’s worth mentioning that the WHILE loop is placed out of the FOR loop, making the FOR loop do not stop at 10 but continuous available instead. Even if the number overtake 10, it doesn’t mean the thread is out of bounds, but the actual times it ran. Let me make some familiar modifications.

- As a further consideration, **we can also rewrite this part of code like shown below**, which satisfy both WHILE loop and thread counting function.

```
int count = 0;
while (true) {
        System.out.println(count%10);
        try {
            socket.receive(packet);
            servicePool.execute(new Response(packet));
        } catch (IOException e) {
            e.printStackTrace();
        }
    count += 1;
}
```

  As it shown above, the “count” variable keeps increasing every time when the inner loop comes to an end. Instead of limiting its upper times, we use count%10 to represent a certain thread is handling a certain event.

****

That’s all. We hope that we have given you a clearer description just now.
Anyway, thanks for your patience, and it will be my pleasure if you read my explanation and modification above. Thanks for that!