Networking	**Programming Project 2**
CSC3935 Spring 2024 (Kissel)	**Due Date: Wednesday Feb. 28, 2024**

A DNS sinkhole is a type of DNS server that will resolve queries to questionable hosts to a preconfigured IP address (most commonly the null address `0.0.0.0` for IPv4). For legitimate hosts, the sinkhole simply passes the queries on to some other (upstream) DNS server and forwards the response to the original requester.

DNS sinkholes have applications in ad blocking and net nanny style filtering, to name a few. Unlike traditional web blockers that operate as extensions, and thus are bound to a specific browser, a DNS sinkhole does filtering for a whole edge network. In this project you will implement a DNS sinkhole in Java. Your sinkhole should allow users to filter both type A and type AAAA queries.

You will write a single application in this project. When your sinkhole starts up it load its blocked hosts list, bind to a port, and wait for queries to arrive. When a query arrives, it should be decoded and checked against the block list. If the hostname and type is found in the block list, a DNS record with the same type and the null address should be returned. Otherwise, the request should be forwarded to a higher level DNS server (e.g., `8.8.8.8`). When a response is received, it should be forwarded to the client.

Preparation

This project will be your first foray in to working with binary protocols. The binary protocol you will work with is DNS (RFC 1035). I'm not expecting you to read the RFC in its entirety, however, you may find it helpful if you want to look up specifics of packet fields. You will also find the DNS section of the Unit 3 lecture notes helpful (pages 8 – 13). As we discussed DNS is a distributed database of sorts where clients are able to ask servers to resolve name questions (most commonly mapping hostnames to IP addresses). This data is stored in a record called a *resource record*. The process consists of a client making a request for all records that match their query, and the server replying with a response that contains all of the matching resource records.

A DNS message consists of a header (12 bytes), a question section (variable size), an answer section (variable size), an authority section (variable size), and an additional information section (variable size). For a request (also called a DNS query) we will only fill in the header and question section leaving all other sections to have size zero. For a reply (also called a DNS answer), we will only fill in the header, question section, and answer section leaving all other sections to have size zero. It is important to note that the reply message should have its question section filled in such that it matches the request's question section.

DNS Header

The DNS header (RFC 1035 §4.1.1) format consists of the following fields in this order:

1. A two byte identifier/transaction ID.
2. A two byte field that represents the flags of the message (this provides details about the message)
3. A two byte field that represents the number of queries in the message.
4. A two byte field that represents the number of answers in the message.
5. A two byte field that represents the number of name server/authority questions.
6. A two byte field that represents the additional information section.

In Java we can represent the header as an object that contains the six fields described above. Since we want each field to be exactly two bytes in size, we need to use the appropriate datatype. In Java, that datatype is `short`. A `short` is nothing more than a two-byte form of an `int` (which is 4 bytes, by the way). The identifier/transaction ID field should be a random number and is used to link the request to the reply (the request creates the ID and the reply contains the ID).The only challenging field to deal with is the flags field. When working with flags we use a process called *bit twiddling*. In bit twiddling we see to *set*, *clear*, or *check* certain bits or sequence of bit in a, in this case, `short` or 16 bits. We perform the operations as follows:

- **Set a Bit (or Bits)**. To set a bit or bits we use the bitwise or operation, denoted `|` in Java. For example, say we want to set bit 3 of the `flags` we would write `flags = flags | 0x0004`. We have encoded our number in a base-16 or hexadecimal. In hexadecimal each digit represent 4 bits. Each of the four bits have a value associated with them. The first bit is 1, the second 2, the third 4, and the fourth 8. If you wish to set multiple bits in one of these four bytes chunks (formally called a nibble) you add up the sum of the bit position values and you get a number between 0 and 15 inclusive. Since we can represent the numbers $10 - 15$ as a single digit, hexadecimal introduces new digits that are letters, $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, and $F = 15$. For example if we wish to set bits 4, 6, and 7 (counting from bit 0 as is traditional) of flags we would write `flags = flags | 0x00E0`. In essence the bitwise or works like the logical or, but it treats 0 as false and 1 as true.
- **Clear a Bit (or Bits)**. To clear a bit or bits we use the bitwise and operation, denoted `&` in Java. For example say we want to clear bit 3 of `flags` we would write `flags = flags & 0xFFFB` observe we have constructed a number where all the bits are one *except* bit 3.

- **Check a Bit (or Bits)**. Imagine we wish to determine if a given flag is set. Let's say we want to check if bit 9 (on a zero count scale) is set. We would write `(flags & 0x0100) == 0x0100`. If you work out the bitwise logic for and, you will see the only way that `flags & 0x0100` equals `0x0100` is if bit 9 is set. We can similarly check if the bit is not set by comparing to `0x0000`. If we want to check multiple bits at the same time, we can do this by constructing the bit string we would use to *set* those corresponding flags.

When building your DNS header object I strongly encourage you to utilize a builder pattern. To see the values of the flags bits for DNS as well as their meaning, please see the description in Section 4.1.1 of RFC 1035.

Question Section

A question is the distinguishing characteristic of a request. The format of a question is described in Section 4.1.2 of RFC 1035. There are three fields:

1. a name which is variable length null byte terminated (which means the last byte is 0x00).

2. a type which is a two byte number that represents a type of request. You will only be concerned with Type A (Hostname to IPv4 address) which is number `0x0001` and Type AAAA (Hostname to IPv6 address) which is number `0x001C`.

3. a (network) class which is a two byte number that represents the type of network this the request is for. For this project, and in pretty much all DNS traffic in the world, the type is `IN`, which is number `0x0001`.

The name field requires a bit more explanation. A domain name is first split into separate strings at the dot `.` and then for each of these substrings we encode the string as a length byte followed by the bytes of the substring. We do this for every substring and append a null byte 0x00 at the end of the last encoded substring. For example, if we apply this process to `cs.merrimack.edu` we would have three substrings `"cs"`, `"merrimack"`, and `"edu"`, the encoded form would be: `0x02 <bytes from cs> 0x09 <bytes from merrimack> 0x03 <bytes from edu> 0x00`.

In the question section should have the same number of questions as the question count specified in the header. Almost always, this is one. If you are sending only a question (i.e., querying a DNS server), the query/response (QR) bit of the flags field should be set to 0.

Answer Section

The answer section is created by a DNS server and contains the responses to a query. Each answer is called a resource record and is formatted as described in Section 4.1.3. The fields are:

1. a name which is stored in the same was a name field in a question.
2. a two byte type field again will will only be concerned with Type `A` and Type `AAAA` records. The constants are the same as in the question section description.
3. A two byte class field, again this will always be `IN`.
4. A four byte (`int`) field denoting the validity time of the resource record, called the *time to live* (TTL).
5. A two byte resource data length `rdlength` that determines the length of the resource data in bytes. In the case of `A` resource records this is 4 and in the case of `AAAA` records this is 16.
6. The resource data `rdlength` bytes of data. In the case of type `A` this is the four bytes of the IP address and in the case of `AAAA` this is the 16 bytes of the IPv6 address.

When you are sending an answer you should be sure that the query/response (QR) bit of the flags field is set to 1.

Serializing Packets

Unlike the plaintext protocols we have seen where we marshaled our objects as JSON, in the case of DNS we wish to marshal our data in binary. When performing either serialization or deserialization you are encouraged to make use of the `ByteBuffer` object. During serialization you will find the `allocate`, `putShort`, `putInt`, and `put` methods helpful. During deserialization you will find the `wrap`, `getShort`, and `get` methods helpful.

Recommended Objects

I recommend that you construct the following objects that should be able to be serialized to the appropriate binary representation as we as deserialized from the appropriate binary representation, unless otherwise noted.

- An object that represents the DNS header.
- An object that represents a query.
- An object that represents a response (**this does not need a decoder, as you will never use it.**)
- An object that represents a type A resource record.
- An object that represents a type AAAA resource record.

You may want to use the builder pattern to help you, I found the builder pattern most helpful for the header and query objects. For the query and response objects I would also add a `buildPacket` method that builds a DNS packet from the information contained and returns the result as a `DatagramPacket`.

The Sinkhole Service

Your sinkhole application should operate as a server. As previously stated this server is responsible for receiving a DNS query over UDP from the client, extracting the question, consulting the block list, and either forwarding the query to the upstream DNS or constructing a null address record of the appropriate type. Finally, the response (null record or upstream DNS) should be forwarded to the client. Since this is a server, this work should be done in an infinite loop.

Traditionally, DNS servers (including sinkholes) listen on UDP port 53. However, this will require administrative access on your computer so you should just pick an available port on your machine (see the testing section for implications of this change). It is also important to remember that when you need to forward a request to an upstream DNS server, you will have to construct a new UDP socket (`DatagramSocket`) for that connection, since in this case you are operating as a client of the upstream DNS server.

In order to operate correctly the configuration information is required. This includes the port number the sinkhole listens on, the location of block list file, and the address of the upstream DNS server

- A field with key `dns-address` that specifies the upstream DNS servers IP address as a string. You should always assume that this server is bound to port 53.
- A field with key `sinkhole-port` that represents the port for the sinkhole service as an integer.
- A field with key `block-file` that specifies the file holding the Type A and Type AAAA queries to block.

By default the configuration information should be read from a file `config.json` *without* user intervention. You are allowed to have an option to your program that allows the user to specify the location of the configuration file but, again the user should not have to.

The block file is a JSON file containing a class with a single field with key `records` whose value is an array of JSON `record` objects. The record object consists of two fields:

- A field with key `host` that represents the host name as a string.
- A field with key `type` that represents the query type, this should be either `"A"` or `"AAAA"`.

Helpful Hints

I offer the following helpful hints as you embark on the project:

- Networking code is notorious for getting complicated fast, be sure to abstract and modularize your code for ease of debugging.

- Employ good design principles.
- Use Wireshark to aid you in debugging your code. If you set Wireshark to look at the loopback interface only you should be able to quickly isolate your protocol messages (remember the will show up in TCP segments). You will also find it helpful to set the protocol to `dns` in the filter bar. Remember in your preferences, under layout, you can turn on "Packet Diagram" to see a graphical representation of your packets. Moreover, if you right click on the packet diagram and click "Show field values" your field values will be placed in your diagram as well.
- Only one `DatagramSocket` can be bound to a port at a time. Remember these port numbers should be on the high end of the numbers (anything 1024 or below is reserved, and probably used by your machine) It is possible that a port can be hung during development. You can correct this by resetting your network card (or just restarting your computer).
- All JSON objects should have a corresponding Java class that implements the `JSONSerializable` interface.
- The Java docs are your friend when working with new classes such as `DatagramPacket`, `DatagramSocket`, and `ByteBuffer`.
- The IP address of your DNS sinkhole is 127.0.0.1 when testing locally.
- There are many potential upstream DNS servers you can use. You could use the on campus DNS servers (only available on the collecge network 10.0.0.3, 10.0.0.4) or you can use a publicly available DNS such as Google's DNS 8.8.8.8 or 8.8.4.4 A list of potential public DNS servers if you don't wish to use Google's can be found at `https://public-dns.info/`.
- You should *not* have to disable your firewall to work on this project. Again, **do not disable your firewall**.
- I've mentioned it earlier but, **start early** and work as a team.
- In order to get the JSON support you need to download or clone `merrimackutil` from `https://github.com/kisselz/merrimackutil`

Testing

To test your application you will want to use the `nslookup` command. First you should start your sinkhole service then you can use the `-port=` command line argument to `nslookup` to change the port used for making DNS queries. For example if I was running my DNS sinkhole on port 5000 on localhost and I wanted to ask about `merrimack.edu` I would write

```
nslookup -port=5000 merrimack.edu 127.0.0.1
```

You can optionally perform a more interesting test by doing the following:

1. Either kill the process that is using UDP port 53 on your machine or on another machine. Call this machine the DNS server. If you wish to allow people to access your sinkhole from other machines you should open UDP port 53 on your firewall.
2. Start up your sinkhole service (being sure its configured to listen to UDP port 53).
3. Open your network settings on your computer and set your DNS server (also called the Name Server in some operating systems) such that it points to the IP address of your DNS server. If you are only running the sinkhole for yourself you can enter 127.0.0.1.
4. Open your browser and try to visit a page the sinkhole will block and one that it should not block and observe the results.

Examples

The sinkhole service has the following basic options.

```
$ java -jar dist/sinkhole.jar -h
usage:
  sinkhole --config <config>
  sinkhole --help
options:
  -c, --config Config file to use.
  -h, --help Display the help.
```

Say I have blocked `google.com` for both type A and AAAA queries in my block list. Then a `nslookup` query for `google.com` would result in:

```
$ nslookup -port=5000 google.com 127.0.0.1
Server: 127.0.0.1
Address: 127.0.0.1#5000

Non-authoritative answer:
Name: google.com
Address: 0.0.0.0
Name: google.com
Address: ::
```

If I query for `merrimack.edu` which is not in the block list, my results would be:

```
$ nslookup -port=5000 merrimack.edu 127.0.0.1
Server: 127.0.0.1
Address: 127.0.0.1#5000

Non-authoritative answer:
Name: merrimack.edu
Address: 23.185.0.3
Name: merrimack.edu
Address: 2620:12a:8001::3
Name: merrimack.edu
Address: 2620:12a:8000::3
```

Source Control

In all projects in this class you will use source control. The source control system the department has chosen is `git`. In particular we will use GitHub. For this project please create a *private* GitHub repository under and account linked to your Merrimack e-mail.

As part of your grade on this project I will be requiring good use of the source control. This means your project should have several commits over the course of time.

Don't develop your entire program outside of source control and then add it to source control before submission!

Every commit should have a well worded commit message that explains what the update to the repository is all about. Please be sure to not put automatically generated files in the repository. This includes `class` files, `jar` files, and `zip` files.

Since this a group project each members contributions will be judged based on the commit history. For example, a group member that has one tiny commit will receive a lower grade than other members. To make your submission clear, please have a `README` in your repository that lists all of the members of the group.

Submitting

To submit this program, please add me as a contributor to your private GitHub repository, my username is (`kisselz@merrimack.edu`). Since this is a group project, there should only be one repository which each group member contributes to.

Rubric

Your grade on this project will be determined as follows:

- Good use of source control (10 points).

- Code is well architected (10 points).
- Configuration file support correctly implemented (5 points).
- The block list is correctly implemented (10 points).
- DNS headers are correctly supported (15 points).
- DNS queries are correctly supported (15 points).
- DNS responses and resource records are correctly supported (20 points)
- Sinkhole algorithm correctly implemented (15 points).