

Computer Networking : Lab1

- Student1's name (email)
- Student2's name (email)

List of materials submitted:

- **Experimental report**
- **Codes**

Checkpoint due : 23:59 pm on October 29th, 2023

Part1: Simple Socket Program (Python, Java, C) (26%)

Please **choose one of the following topics** to complete.

Score points:

- **Program correctness (20%)**
- **Screenshot of code running & code description (6%)**
- **TCP Socket Program**

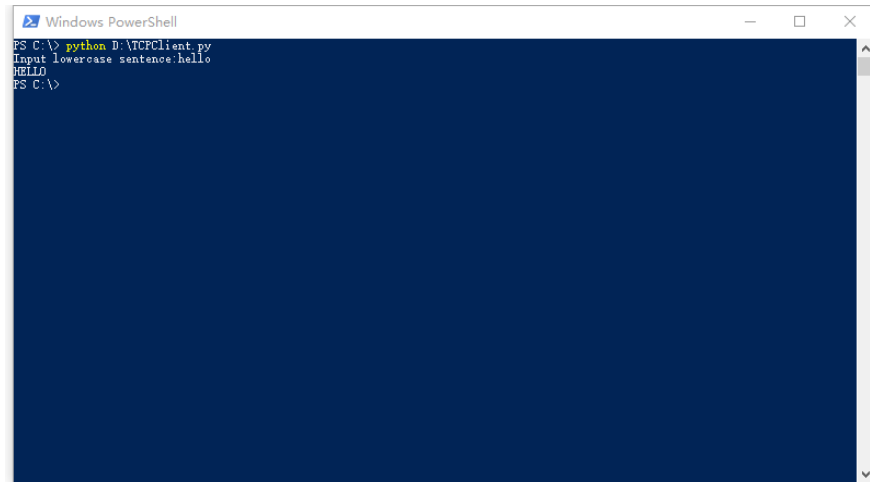
Question requirements:

The client program creates a TCP socket, then initiates a connection to the specified server address and port, waits for the server to connect, then sends the string entered by the user through the socket, and then displays the message returned by the server.

The server program always maintains a TCP welcome socket and can receive connection requests from any client. After receiving the client's connection request, create a new TCP connection socket for communicating with the client alone, while displaying the client address and port. After receiving the string sent by the client, change it to uppercase, and then return the modified string to the client. Finally, close the TCP connection socket.

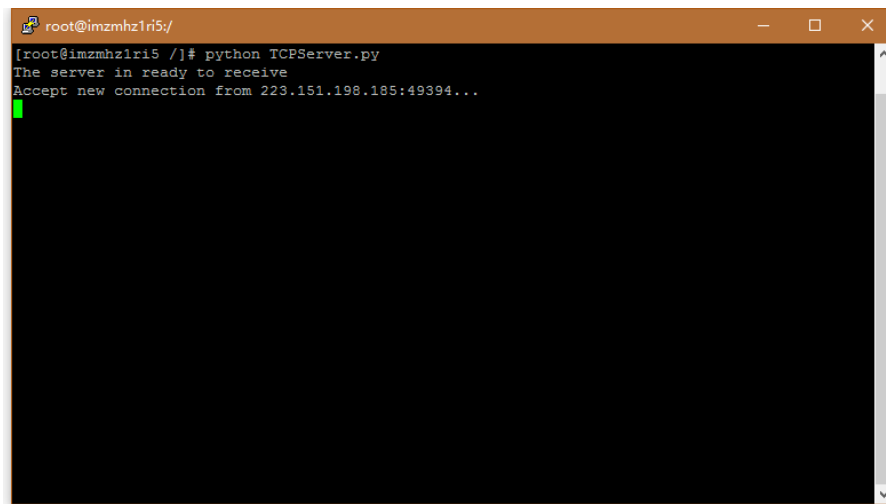
Running result:

- Client



```
Windows PowerShell
FS C:\> python D:\TCPClient.py
Input lowercase sentence:hello
HELLO
FS C:\>
```

■ Server



```
root@imzmhz1ri5:/
[root@imzmhz1ri5 /]# python TCPServer.py
The server is ready to receive
Accept new connection from 223.151.198.185:49394...
```

• UDP Socket Program

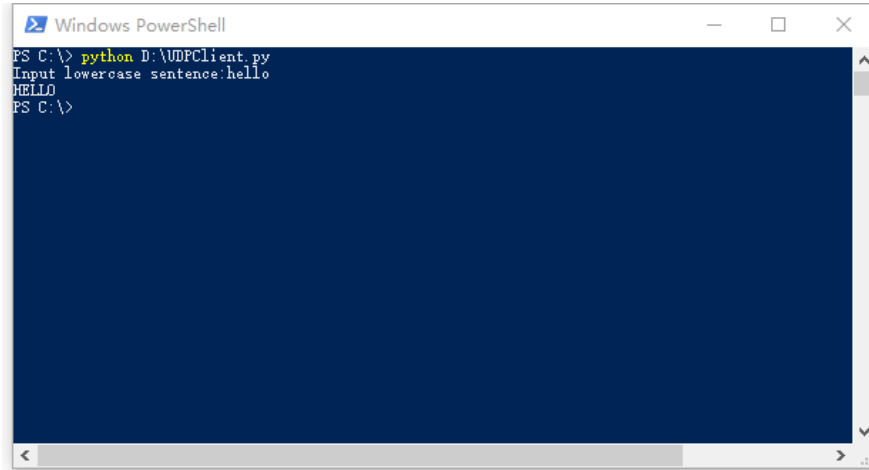
Question requirements:

The client program creates a UDP socket and sends it to the specified server address and corresponding port after the user inputs a string of lowercase letters. It waits for the server to return a message and then displays the message.

The server program always maintains a connectable UDP socket. After receiving the string, it changes it to uppercase and then returns the modified string to the client.

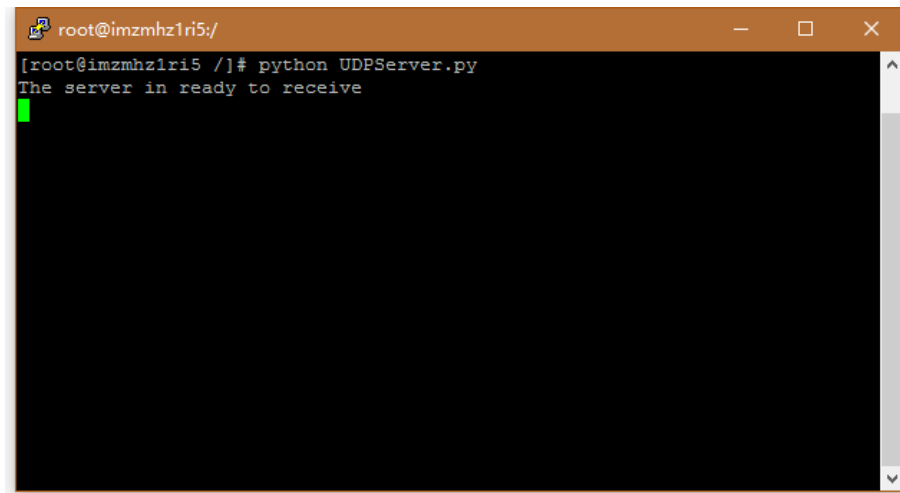
Running result:

■ Client



```
Windows PowerShell
PS C:\> python D:\UDPClient.py
Input lowercase sentence:hello
HELLO
PS C:\>
```

■ Server



```
root@imzmhz1ri5:/
[root@imzmhz1ri5 /]# python UDPServer.py
The server is ready to receive
```

Part2: HTTP

Having gotten our feet wet with the Wireshark packet sniffer in the introductory lab, we're now ready to use Wireshark to investigate protocols in operation. In this lab, we'll explore several aspects of the HTTP protocol: the basic GET/response interaction, HTTP message formats, retrieving large HTML files, retrieving HTML files with embedded objects, and HTTP authentication and security. Before beginning these labs, you might want to review Section 2.2 of the text.

Score points:

- Small question (1.5% * 19)
- Experiment screenshot & coherence, logic and simplicity of text description (10%)

- The Basic HTTP GET/response interaction

Let's begin our exploration of HTTP by downloading a very simple HTML file - one that is very short, and contains no embedded objects. Do the following:

1. Start up your web browser.
2. Start up the Wireshark packet sniffer, as described in the Introductory lab (but don't yet begin packet capture). Enter "http" (just the letters, not the quotation marks, and in lower case) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window. (We're only interested in the HTTP protocol here, and don't want to see the clutter of all captured packets).
3. Wait a bit more than one minute (we'll see why shortly), and then begin Wireshark packet capture.
4. Enter the following to your browser
<http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html>
 Your browser should display the very simple, one-line HTML file.
5. Stop Wireshark packet capture.

Your Wireshark window should look similar to the window shown in Figure 1. If you're unable to run Wireshark on a live network connection, you can download a packet trace that was created when the steps above were followed.

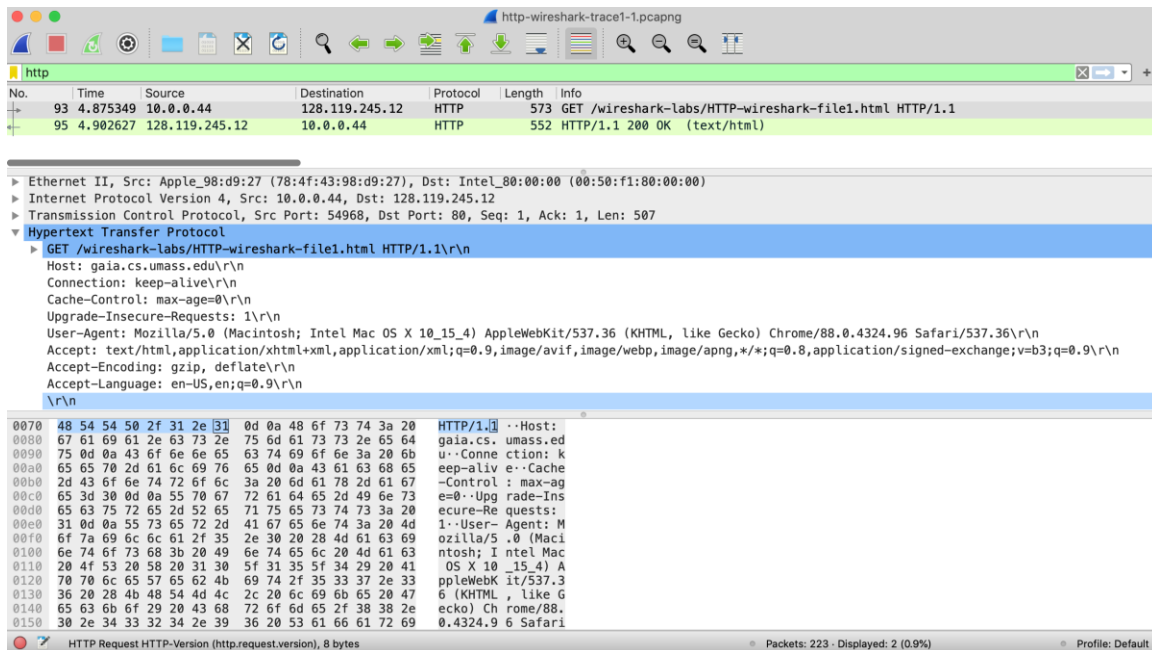


Figure 1: Wireshark Display after <http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file1.html> has been retrieved by your browser

The example in Figure 1 shows in the packet-listing window that two HTTP messages were captured: the GET message (from your browser to the gaia.cs.umass.edu web server) and the response message from the server to your browser. The packet-contents window

shows details of the selected message (in this case the HTTP OK message, which is highlighted in the packet-listing window). Recall that since the HTTP message was carried inside a TCP segment, which was carried inside an IP datagram, which was carried within an Ethernet frame, Wireshark displays the Frame, Ethernet, IP, and TCP packet information as well. We want to minimize the amount of non-HTTP data displayed (we're interested in HTTP here, and will be investigating these other protocols in later labs), so make sure the boxes at the far left of the Frame, Ethernet, IP and TCP information have a plus sign or a right-pointing triangle (which means there is hidden, undisplayed information), and the HTTP line has a minus sign or a down-pointing triangle (which means that all information about the HTTP message is displayed).

(Note: You should ignore any HTTP GET and response for favicon.ico. If you see a reference to this file, it is your browser automatically asking the server if it (the server) has a small icon file that should be displayed next to the displayed URL in your browser. We'll ignore references to this pesky file in this lab.)

By looking at the information in the HTTP GET and response messages, answer the following questions.

1. Is your browser running HTTP version 1.0, 1.1, or 2? What version of HTTP is the server running?
2. What languages (if any) does your browser indicate that it can accept to the server?
3. What is the IP address of your computer? What is the IP address of the gaia.cs.umass.edu server?
4. What is the status code returned from the server to your browser?
5. When was the HTML file that you are retrieving last modified at the server?
6. How many bytes of content are being returned to your browser?
7. By inspecting the raw data in the packet content window, do you see any headers within the data that are not displayed in the packet-listing window? If so, name one.

In your answer to question 5 above (assuming you're running Wireshark "live", as opposed to using an earlier-recorded trace file), you might have been surprised to find that the document you just retrieved was last modified within a minute before you downloaded the document. That's because (for this particular file), the gaia.cs.umass.edu server is setting the file's last-modified time to be the current time, and is doing so once per minute. Thus, if you wait a minute between accesses, the file will appear to have been recently modified, and hence your browser will download a "new" copy of the document.

• The HTTP CONDITIONAL GET/response interaction

Recall from Section 2.2.5 of the text, that most web browsers perform object caching and thus often perform a conditional GET when retrieving an HTTP object. Before performing the steps below, make sure your browser's cache is empty. Now do the following:

- Start up your web browser, and make sure your browser's cache is cleared, as discussed above.

- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
<http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file2.html>
Your browser should display a very simple five-line HTML file.
- Quickly enter the same URL into your browser again (or simply select the refresh button on your browser)
- Stop Wireshark packet capture, and enter “http” (again, in lower case without the quotation marks) in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window.

If you’re unable to run Wireshark on a live network connection (or unable to get your browser to issue an If-Modified-Since field on the second HTTP GET request), you can download a packet trace that was created when the steps above were followed. Answer the following questions:

8. Inspect the contents of the first HTTP GET request from your browser to the server. Do you see an “IF-MODIFIED-SINCE” line in the HTTP GET?
9. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?
10. Now inspect the contents of the second HTTP GET request from your browser to the server. Do you see an “IF-MODIFIED-SINCE:” line in the HTTP GET? If so, what information follows the “IF-MODIFIED-SINCE:” header?
11. What is the HTTP status code and phrase returned from the server in response to this second HTTP GET? Did the server explicitly return the contents of the file? Explain.

• Retrieving Long Documents

In our examples thus far, the documents retrieved have been simple and short HTML files. Let’s next see what happens when we download a long HTML file. Do the following:

- Start up your web browser, and make sure your browser’s cache is cleared, as discussed above.
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
<http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file3.html>
Your browser should display the rather lengthy US Bill of Rights.
- Stop Wireshark packet capture, and enter “http” in the display-filter-specification window, so that only captured HTTP messages will be displayed.

In the packet-listing window, you should see your HTTP GET message, followed by a multiple-packet TCP response to your HTTP GET request. Make sure your Wireshark display filter is cleared so that the multi-packet TCP response will be displayed in the packet listing.

This multiple-packet response deserves a bit of explanation. Recall from Section 2.2 (see Figure 2.9 in the text) that the HTTP response message consists of a status line, followed by header lines, followed by a blank line, followed by the entity body. In the case of our

HTTP GET, the entity body in the response is the *entire* requested HTML file. In our case here, the HTML file is rather long, and at 4500 bytes is too large to fit in one TCP packet. The single HTTP response message is thus broken into several pieces by TCP, with each piece being contained within a separate TCP segment (see Figure 1.24 in the text). In recent versions of Wireshark, Wireshark indicates each TCP segment as a separate packet, and the fact that the single HTTP response was fragmented across multiple TCP packets is indicated by the “TCP segment of a reassembled PDU” in the Info column of the Wireshark display.

Answer the following questions:

12. How many HTTP GET request messages did your browser send? Which packet number in the trace contains the GET message for the Bill of Rights?
13. Which packet number in the trace contains the status code and phrase associated with the response to the HTTP GET request?
14. What is the status code and phrase in the response?
15. How many data-containing TCP segments were needed to carry the single HTTP response and the text of the Bill of Rights?

• HTML Documents with Embedded Objects

Now that we’ve seen how Wireshark displays the captured packet traffic for large HTML files, we can look at what happens when your browser downloads a file with embedded objects, i.e., a file that includes other objects (in the example below, image files) that are stored on another server(s).

Do the following:

- Start up your web browser, and make sure your browser’s cache is cleared, as discussed above.
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
<http://gaia.cs.umass.edu/wireshark-labs/HTTP-wireshark-file4.html>
Your browser should display a short HTML file with two images. These two images are referenced in the base HTML file. That is, the images themselves are not contained in the HTML; instead the URLs for the images are contained in the downloaded HTML file.
- Stop Wireshark packet capture, and enter “http” in the display-filter-specification window, so that only captured HTTP messages will be displayed.

Answer the following questions:

16. How many HTTP GET request messages did your browser send? To which Internet addresses were these GET requests sent?
17. Can you tell whether your browser downloaded the two images serially, or whether they were downloaded from the two web sites in parallel? Explain.

- HTTP Authentication

Finally, let's try visiting a web site that is password-protected and examine the sequence of HTTP message exchanged for such a site. The URL http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html is password protected. The username is "wireshark-students" (without the quotes), and the password is "network" (again, without the quotes). So let's access this "secure" password-protected site. Do the following:

- Make sure your browser's cache is cleared, as discussed above, and close down your browser. Then, start up your browser
- Start up the Wireshark packet sniffer
- Enter the following URL into your browser
http://gaia.cs.umass.edu/wireshark-labs/protected_pages/HTTP-wireshark-file5.html
Type the requested user name and password into the pop up box.
- Stop Wireshark packet capture, and enter "http" in the display-filter-specification window, so that only captured HTTP messages will be displayed later in the packet-listing window.

Answer the following questions:

18. What is the server's response (status code and phrase) in response to the initial HTTP GET message from your browser?
19. When your browser's sends the HTTP GET message for the second time, what new field is included in the HTTP GET message?

The username (wireshark-students) and password (network) that you entered are encoded in the string of characters (d2lyZXNoYXJrLXN0dWRlbnRzOm5ldHdvcm0=) following the "Authorization: Basic" header in the client's HTTP GET message. While it may appear that your username and password are encrypted, they are simply encoded in a format known as Base64 format. The username and password are *not* encrypted! To see this, go to <http://www.motobit.com/util/base64-decoder-encoder.asp> and enter the base64-encoded string d2lyZXNoYXJrLXN0dWRlbnRz and decode. *Voila!* You have translated from Base64 encoding to ASCII encoding, and thus should see your username! To view the password, enter the remainder of the string Om5ldHdvcm0= and press decode. Since anyone can download a tool like Wireshark and sniff packets (not just their own) passing by their network adaptor, and anyone can translate from Base64 to ASCII (you just did it!), it should be clear to you that simple passwords on WWW sites are not secure unless additional measures are taken.

Fear not! As we will see in Chapter 8, there are ways to make WWW access more secure. However, we'll clearly need something that goes beyond the basic HTTP authentication framework!

Part3: DNS

As described in Section 2.4 of the text, the Domain Name System (DNS) translates hostnames to IP addresses, fulfilling a critical role in the Internet infrastructure. In this lab, we'll take a closer look at the client side of DNS. Recall that the client's role in the DNS is relatively simple – a client sends a *query* to its local DNS server, and receives a *response* back. As shown in Figures 2.19 and 2.20 in the textbook, much can go on “under the covers,” invisible to a DNS client, as the hierarchical DNS servers communicate with each other to either recursively or iteratively resolve the client's DNS query. From the DNS client's standpoint, however, the protocol is quite simple – a query is formulated to the local DNS server and a response is received from that server.

Before beginning this lab, you'll probably want to review DNS by reading Section 2.4 of the text. In particular, you may want to review the material on **local DNS servers**, **DNS caching**, **DNS records and messages**, and the **TYPE field** in the DNS record.

Score points:

- **Small question (1.5% * 17)**
- **Experiment screenshot & coherence, logic and simplicity of text description (10%)**
- **nslookup**

Let's start our investigation of the DNS by examining the `nslookup` command, which will invoke the underlying DNS services to implement its functionality. The `nslookup` command is available in most Microsoft, Apple IOS, and Linux operating systems. To run `nslookup` you just type the `nslookup` command on the command line in a DOS window, Mac IOS terminal window, or Linux shell.

In its most basic operation, `nslookup` allows the host running `nslookup` to query any specified DNS server for a DNS record. The queried DNS server can be a root DNS server, a top-level-domain (TLD) DNS server, an authoritative DNS server, or an intermediate DNS server (see the textbook for definitions of these terms). For example, `nslookup` can be used to retrieve a “Type=A” DNS record that maps a hostname to its IP address. To accomplish this task, `nslookup` sends a DNS query to the specified DNS server (or the default local DNS server for the host on which `nslookup` is run, if no specific DNS server is specified), receives a DNS response from that DNS server, and displays the result.

Let's take `nslookup` out for a spin! We'll first run `nslookup` on the Linux command line on the `www.fudan.edu.cn` host located at Fudan University.

Comand: nslookup www.fudan.edu.cn

In addition to using `nslookup` to query for a DNS “Type=A” record, we can also use `nslookup` to query for a “TYPE=NS” record, which returns the hostname

(and its IP address) of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the authoritative server's domain.

Invoking `nslookup` with the option “-type=NS” and the domain “fudan.edu.cn” causes `nslookup` to send a query for a type-NS record to the default local DNS server. In words, the query is saying, “please send me the host names of the authoritative DNS for fudan.edu.cn”. (When the `-type` option is not used, `nslookup` uses the default, which is to query for type A records.)

`nslookup` has a number of additional options beyond “-type=NS” that you might want to explore. Here's a site with screenshots of ten popular `nslookup` uses: <https://www.cloudns.net/blog/10-most-used-nslookup-commands/> and here are the “man pages” for `nslookup`: <https://linux.die.net/man/1/nslookup>.

Lastly, we sometimes might be interested in discovering the name of the host associated with a given IP address. `nslookup` can also be used to perform this so-called “reverse DNS lookup.” For example, we can specify an IP address as the `nslookup` argument and `nslookup` returns the host name with that address.

Now that we've provided an overview of `nslookup`, it's time for you to test drive it yourself. Do the following (and write down the results).

1. Run `nslookup` to obtain the IP address of the web server for Fudan University: <https://www.fudan.edu.cn>. What is the IP address of fudan.edu.cn?
2. What is the IP address of the DNS server that provided the answer to your `nslookup` command in question 1 above?
3. Did the answer to your `nslookup` command in question 1 above come from an authoritative or non-authoritative server?
4. Use the `nslookup` command to determine the name of the authoritative name server for the fudan.edu.cn. What is that name? (If there are more than one authoritative servers, what is the name of the first authoritative server returned by `nslookup`)? If you had to find the IP address of that authoritative name server, how would you do so?

- The DNS cache on your computer

From the description of iterative and recursive DNS query resolution (Figures 2.19 and 2.20) in our textbook, you might think that the local DNS server must be contacted *every* time an application needs to translate from a hostname to an IP address. That's not always true in practice!

Most hosts (e.g., your personal computer) keep a *cache* of recently retrieved DNS records (sometimes called a DNS *resolver cache*), just like many Web browsers keep a cache of objects recently retrieved by HTTP. When DNS services need to be invoked by a host, that host will first check if the DNS record needed is resident in this host's DNS cache; if the record is found, the host will not even bother to contact the local DNS server and will instead use this cached DNS record. A DNS record in a resolver cache will eventually

timeout and be removed from the resolver cache, just as records cached in a local DNS server (see Figures 2.19, 2.20) will timeout.

You can also explicitly clear the records in your DNS cache. There's no harm in doing so – it will just mean that your computer will need to invoke the distributed DNS service next time it needs to use the DNS name resolution service, since it will find no records in the cache. On a Mac computer, you can enter the following command into a terminal window to clear your DNS resolver cache:

```
sudo killall -HUP mDNSResponder
```

On Windows computer you can enter the following command at the command prompt:

```
ipconfig /flushdns
```

and on a Linux computer, enter:

```
sudo systemd-resolve --flush-caches
```

- **Tracing DNS with Wireshark**

Now that we are familiar with `nslookup` and clearing the DNS resolver cache, we're ready to get down to some serious business. Let's first capture the DNS messages that are generated by ordinary Web-surfing activity.

- Clear the DNS cache in your host, as described above.
- Open your Web browser and clear your browser cache.
- Open Wireshark and enter `ip.addr == <your_IP_address>` into the display filter, where `<your_IP_address>` is the IPv4 address of your computer. With this filter, Wireshark will only display packets that either originate from, or are destined to, your host.
- Start packet capture in Wireshark.
- With your browser, visit the Web page: <https://www.fudan.edu.cn/>
- Stop packet capture.

Answer the following questions.

5. Locate the first DNS query message resolving the name `fudan.edu.cn`. What is the packet number in the trace for the DNS query message? Is this query message sent over UDP or TCP?
6. Now locate the corresponding DNS response to the initial DNS query. What is the packet number in the trace for the DNS response message? Is this response message received via UDP or TCP?
7. What is the destination port for the DNS query message? What is the source port of the DNS response message?
8. To what IP address is the DNS query message sent?
9. Examine the DNS query message. How many “questions” does this DNS message contain? How many “answers” answers does it contain?
10. Examine the DNS response message to the initial query message. How many “questions” does this DNS message contain? How many “answers” answers does it contain?

Now let's play with `nslookup`.

- Start packet capture.
- Do an `nslookup` on `www.fudan.edu.cn`
- Stop packet capture.

You should get a trace that looks something like the following in your Wireshark window. Let's look at the first type **A** query (indicated by the "A" in the *Info* column for that packet).

11. What is the destination port for the DNS query message? What is the source port of the DNS response message?
12. To what IP address is the DNS query message sent? Is this the IP address of your default local DNS server?
13. Examine the DNS query message. What "Type" of DNS query is it? Does the query message contain any "answers"?
14. Examine the DNS response message to the query message. How many "questions" does this DNS response message contain? How many "answers"?

Last, let's use `nslookup` to issue a command that will return a type NS DNS record, Enter the following command:

```
nslookup -type=NS fudan.edu.cn
```

and then answer the following questions :

15. To what IP address is the DNS query message sent? Is this the IP address of your default local DNS server?
16. Examine the DNS query message. How many questions does the query have? Does the query message contain any "answers"?
17. Examine the DNS response message. How many answers does the response have? What information is contained in the answers? How many additional resource records are returned? What additional information is included in these additional resource records?