CSE434 Computer Networks (FALL, 2009)
Programming Assignment 2
Due: Wed, October 7, 2009

Submission Procedure: No late submissions will be accepted. Submit a softcopy before the class to su.kim.asu@gmail.com. To prove your submission, you MUST keep a copy of your sent email until you get your grade.

Deliverables: Create a folder named "YourName_PA2" (e.g., SuKim_PA2). Copy all your source files for Task 1 and 2 and a report for Task 3 and 4 into this folder. Zip the folder. Make sure the file name is "YourName_PA2.zip."

This assignment is to develop a UDP ping client and a web proxy server in JAVA and study HTTP and DNS using Wireshark. Before you start, you need to download a package (PA2-package.zip) and unzip it for Task 1 and 2.

For students who are familiar with JAVA socket programming, we expect to complete Task 1 and 2 within 12 hours including test. For others, they can take a couple of days.

For Task 3 and 4, playing Wireshark, collecting/analyzing data, and writing a report will take about 12 hours. Note that Task 3 has 19 questions and Task 4 has 14 questions. Also, there might be a problem to connect a server at a particular time. Do not wait until the last day!

Task 1: UDP Ping

In this part, you will study a simple Internet ping server written in the Java language, and implement a corresponding client. The functionalities provided by these programs are similar to the standard ping programs available in modern operating systems, except that they use UDP rather than Internet Control Message Protocol (ICMP) to communicate with each other. (Java does not provide a straightforward means to interact with ICMP.)

The ping protocol allows a client machine to send a packet of data to a remote machine, and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines.

You are given the complete code for the Ping server below. Your job is to write the Ping client.

Server Code

Open "PingServer.java". You should study this code carefully, as it will help you write your Ping client. The server sits in an infinite loop listening for incoming UDP packets. When a packet comes in, the server simply sends the encapsulated data back to the client.

Packet Loss

UDP provides applications with an unreliable transport service, because messages may get lost in the network due to router queue overflows or other reasons. In contrast, TCP provides applications with a reliable transport service and takes care of any lost packets by retransmitting them until they are successfully received. Applications using UDP for communication must therefore implement any reliability they need separately in the application level (each application can implement a different policy, according to its specific needs).

Because packet loss is rare or even non-existent in typical campus networks, the server in this lab injects artificial loss to simulate the effects of network packet loss. The server has a parameter LOSS_RATE that determines which percentage of packets should be lost.

The server also has another parameter AVERAGE_DELAY that is used to simulate transmission delay from sending a packet across the Internet. You should set AVERAGE_DELAY to a positive value when testing your client and server on the same machine, or when machines are close by on the network. You can set AVERAGE_DELAY to 0 to find out the true round trip times of your packets.

Compiling and Running Server

To compile the server, do the following:

```
javac PingServer.java
```

To run the server, do the following:

```
java PingServer port
```

where port is the port number the server listens on. Remember that you have to pick a port number greater than 1024, because only processes running with root (administrator) privilege can bind to ports less than 1024.

Note: if you get a class not found error when running the above command, then you may need to tell Java to look in the current directory in order to resolve class references. In this case, the commands will be as follows:

```
java -classpath . PingServer port
```

Your Job: The Client

You should write the client so that it sends 10 ping requests to the server, separated by approximately one second. Each message contains a payload of data that includes the keyword PING, a sequence number, and a timestamp. After sending each packet, the client waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client assumes that its packet or the server's reply packet has been lost in the network.

Hint: Cut and paste PingServer, rename the code PingClient, and then modify the code.

You should write the client so that it starts with the following command:

```
java PingClient host port
```

where host is the name of the computer the server is running on and port is the port number it is listening to. Note that you can run the client and server either on different machines or on the same machine.

The client should send 10 pings to the server. Because UDP is an unreliable protocol, some of the packets sent to the server may be lost, or some of the packets sent from server to client may be lost. For this reason, the client can not wait indefinitely for a reply to a ping message. You should have the client wait up to one second for a reply; if no reply is received, then the client should assume that the packet was lost during transmission across the network. You will need to research the API for DatagramSocket to find out how to set the timeout value on a datagram socket.

When developing your code, you should run the ping server on your machine, and test your client by sending packets to localhost (or, 127.0.0.1). After you have fully debugged your code, you should see how your application communicates across the network with a ping server run by another member of the class.

Message Format

The ping messages in this lab are formatted in a simple way. Each message contains a sequence of characters terminated by a carriage return character (r) and a line feed character (n). The message contains the following string:

```
PING sequence_number time CRLF
```

where sequence_number starts at 0 and progresses to 9 for each successive ping message sent by the client, time is the time when the client sent the message, and CRLF represent the carriage return and line feed characters that terminate the line.

Output Format

Your client program should display the average round-trip time (RTT) of packets and the number of packet losses.

Task 2: Web Proxy Server

In this part, you will develop a small web proxy server which is also able to cache web pages. This is a very simple proxy server which only understands simple GET-requests, but is able to handle all kinds of objects, not just HTML pages, but also images.

Source Codes

The code is divided into three classes as follows:

- ProxyCache.java holds the start-up code for the proxy and code for handling the requests.
- HttpRequest.java contains the routines for parsing and processing the incoming requests from clients.
- HttpResponse.java takes care of reading the replies from servers and processing them.

Your work will be to complete the proxy so that it is able to receive requests, forward them, read replies, and return those to the clients. You will need to complete the classes ProxyCache, HttpRequest, and HttpResponse. The places where you need to fill in code are marked with /* Fill in */. Each place may require one or more lines of code.

NOTE: As explained below, the proxy uses DataInputStreams for processing the replies from servers. This is because the replies are a mixture of textual and binary data and the only input streams in Java which allow treating both at the same time are DataInputStreams. To get the code to compile, you must use the -deprecation argument for the compiler as follows:

```
javac -deprecation *.java
```

If you do not use the -deprecation flag, the compiler will refuse to compile your code!

Running the Proxy

Running the proxy is as follows:

```
java ProxyCache port
```

where *port* is the port number on which you want the proxy to listen for incoming connections from clients.

Configuring Your Browser

You will also need to configure your web browser to use your proxy. This depends on your browser. In Internet Explorer, you can set the proxy in "Internet Options" in the Connections tab under LAN Settings. In Netscape (and derived browsers, such as Mozilla), you can set the proxy in Edit->Preferences and then select Advanced and Proxies.

In both cases you need to give the address of the proxy and the port number which you gave when you started the proxy. You can run the proxy and browser on the same computer without any problems.

Proxy Functionality

The proxy works as follows.

1. The proxy listens for requests from clients
2. When there is a request, the proxy spawns a new thread for handling the request and creates an HttpRequest-object which contains the request.
3. The new thread sends the request to the server and reads the server's reply into an HttpResponse-object.
4. The thread sends the response back to the requesting client.

Your task is to complete the code which handles the above process. Most of the error handling in the proxy is very simple and it does not inform the client about errors. When there are errors, the proxy will simply stop processing the request and the client will eventually get a timeout.

Some browsers also send their requests one at a time, without using parallel connections. Especially in pages with lot of inlined images, this may cause the page to load very slowly.

Caching (Optional)

Caching the responses in the proxy is left as an optional exercise, since it demands a significant amount of additional work. The basic functionality of caching goes as follows.

1. When the proxy gets a request, it checks if the requested object is cached, and if yes, then returns the object from the cache, without contacting the server.
2. If the object is not cached, the proxy retrieves the object from the server, returns it to the client, and caches a copy for future requests.

In practice, the proxy must verify that the cached responses are still valid and that they are the correct response to the client's request. You can read more about caching and how it is handled in HTTP in RFC 2068. For this lab, it is sufficient to implement the above simple policy.

Programming Hints

Most of the code you need to write relates to processing HTTP requests and responses as well as handling Java sockets.

One point worth noting is the processing of replies from the server. In an HTTP response, the headers are sent as ASCII lines, separated by CRLF character sequences. The headers are followed by an empty line and the response body, which can be binary data in the case of images, for example.

Java separates the input streams according to whether they are text-based or binary, which presents a small problem in this case. Only DataInputStreams are able to handle both text and binary data simultaneously; all other streams are either pure text (e.g., BufferedReader), or pure binary (e.g., BufferedInputStream), and mixing them on the same socket does not generally work.

The DataInputStream has a small gotcha, because it is not able to guarantee that the data it reads can be correctly converted to the correct characters on every platform (DataInputStream.readLine() function). In the case of this lab, the conversion usually works, but the compiler will flag the DataInputStream.readLine()-method as deprecated and will refuse to compile without the -deprecation flag.

It is highly recommended that you use the DataInputStream for reading the response.

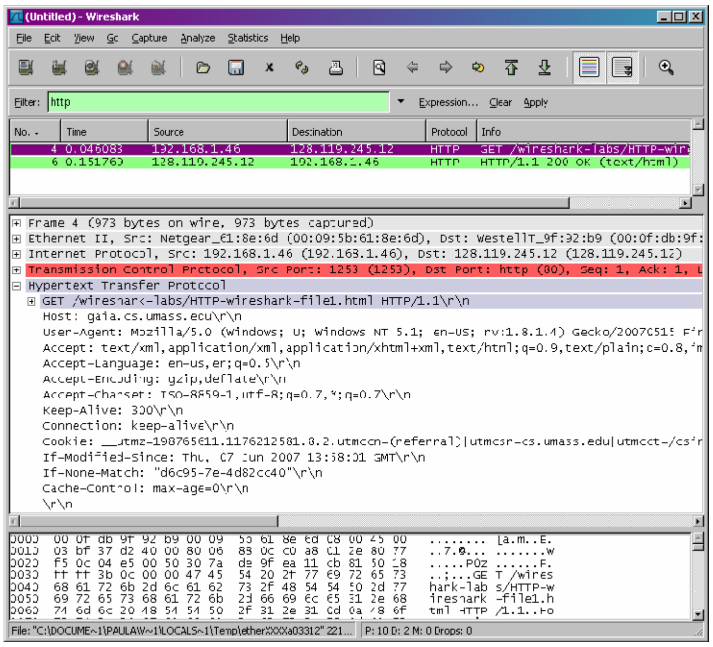
Task 3: Wireshark Lab - HTTP

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.

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) 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.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: http
Expression... Clear Apply

```
No. -  Time       Source          Destination     Protocol  Info
   4  0.046083   192.168.1.46    128.119.245.12  HTTP      GET /wireshark-labs/HTTP-wir
   6  0.151760   128.119.245.12  192.168.1.46    HTTP      HTTP/1.1 200 OK (text/html)

+ Frame 4 (973 bytes on wire, 973 bytes captured)
+ Ethernet II, Src: Netgear_61:8e:6d (00:09:5b:61:8e:6d), Dst: WestellT_9f:92:b9 (00:0f:db:9f:
+ Internet Protocol, Src: 192.168.1.46 (192.168.1.46), Dst: 128.119.245.12 (128.119.245.12)
+ Transmission Control Protocol, Src Port: 1253 (1253), Dst Port: http (80), Seq: 1, Ack: 1, L
- Hypertext Transfer Protocol
  + GET /wireshark-labs/HTTP-wireshark-file1.html HTTP/1.1\r\n
    Host: gaia.cs.umass.edu\r\n
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.4) Gecko/20070515 Fir
    Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,im
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip,deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Keep-Alive: 300\r\n
    Connection: keep-alive\r\n
    Cookie: __utmz=198765611.1176212581.8.2.utmccn=(referral)|utmcsr=cs.umass.edu|utmcct=/csir
    If-Modified-Since: Thu, 07 Jun 2007 13:58:01 GMT\r\n
    If-None-Match: "d6c95-7e-4d82cc40"\r\n
    Cache-Control: max-age=0\r\n
    \r\n

0000  00 0f db 9f 92 b9 00 09  5b 61 8e 6d 08 00 45 00   ........ [a.m..E.
0010  03 bf 37 d2 40 00 80 06  83 0c c0 a8 01 2e 80 77   ..7.@... .......w
0020  f5 0c 04 e5 00 50 30 7a  de 9f ea 11 cb 81 50 18   .....P0z ......P.
0030  ff ff 3b 0c 00 00 47 45  54 20 2f 77 69 72 65 73   ..;...GE T /wires
0040  68 61 72 6b 2d 6c 61 62  73 2f 48 54 54 50 2d 77   hark-lab s/HTTP-w
0050  69 72 65 73 68 61 72 6b  2d 66 69 6c 65 31 2e 68   ireshark -file1.h
0060  74 6d 6c 20 48 54 54 50  2f 31 2e 31 0d 0a 48 6f   tml HTTP /1.1..Ho

File: "C:\DOCUME~1\PAULAW~1\LOCALS~1\Temp\etherXXXXa03312" 221...   P: 10 D: 2 M: 0 Drops: 0
```

Figure 1: Wireshark Display after http://gaia.cs.umass.edu/wireshark-labs/ HTTPwireshark-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 GET 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 is later labs), so make sure the boxes at the far left of the Frame, Ethernet, IP and TCP information have a plus sign (which means there is hidden, undisplayed information), and the HTTP line has a minus sign (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. When answering the following questions, you should print out the GET and response messages to a text file (see the introductory Wireshark lab for an explanation of how to do this) and indicate (highlight) where in the message you've found the information that answers the following questions.

1. Is your browser running HTTP version 1.0 or 1.1? 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? 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, 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 2.2.6 of the text, that most web browsers perform object caching and thus perform a conditional GET when retrieving an HTTP object. Before performing the steps below, make sure your browser's cache is empty. (To do this under Netscape 7.0, select *Edit->Preferences->Advanced->Cache* and clear the memory and disk cache. For Firefox, select *Tools->Clear Private Data*, or for Internet Explorer, select *Tools->Internet Options->Delete File;* these actions will remove cached files from your browser's cache.) 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" 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:

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 response to your HTTP GET request. 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 1.20 in the text). Each TCP segment is recorded as a separate packet by Wireshark, and the fact that the single HTTP response was fragmented across multiple TCP packets is indicated by the "Continuation" phrase displayed by Wireshark. We stress here that there is no "Continuation" message in HTTP!

Answer the following questions:

12. How many HTTP GET request messages were sent by your browser.
13. How many data-containing TCP segments were needed to carry the single HTTP response?
14. What is the status code and phrase associated with the response to the HTTP GET request?

15. Are there any HTTP status lines in the transmitted data associated with a TCP induced "Continuation"?

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. As discussed in the textbook, your browser will have to retrieve these logos from the indicated web sites. Our publisher's logo is retrieved from the www.aw-bc.com web site. The image of our book's cover is stored at the manic.cs.umass.edu server.
- 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 were sent by your browser? 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-wiresharkfile5.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.

Now let's examine the Wireshark output. You might want to first read up on HTTP authentication by reviewing the easy-to-read material on "HTTP Access Authentication Framework" at http://frontier.userland.com/stories/storyReader$2159
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 (wirehsark-students) and password (network) that you entered are encoded in the string of characters (d2lyZXNoYXJrLXN0dWRlbnRzOm5ldHdvcms=) 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.securitystats.com/tools/base64.php and enter the base64-encoded string d2lyZXNoYXJrLXN0dWRlbnRz and press 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 Om5ldHdvcms= 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!

Task 4: Wireshark Lab - DNS

As described in Section 2.5 of the textbook, 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 2.21 and 2.22 in the textbook, much can go on "under the covers," invisible to the DNS clients, 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 2.5 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.

nslookup

In this lab, we'll make extensive use of the *nslookup* tool, which is available in most Linux/Unix and Microsoft platforms today. To run *nslookup* in Linux/Unix, you just type the *nslookup* command on the command line. To run it in Windows, open the Command Prompt and run *nslookup* on the command line.
In it is most basic operation, *nslookup* tool allows the host running the tool to query any specified DNS server for a DNS record. The queried DNS server can be a root DNS server, a top-level-domain DNS server, an authoritative DNS server, or an intermediate DNS server (see the textbook for definitions of these terms). To accomplish this task, *nslookup* sends a DNS query to the specified DNS server, receives a DNS reply from that same DNS server, and displays the result.

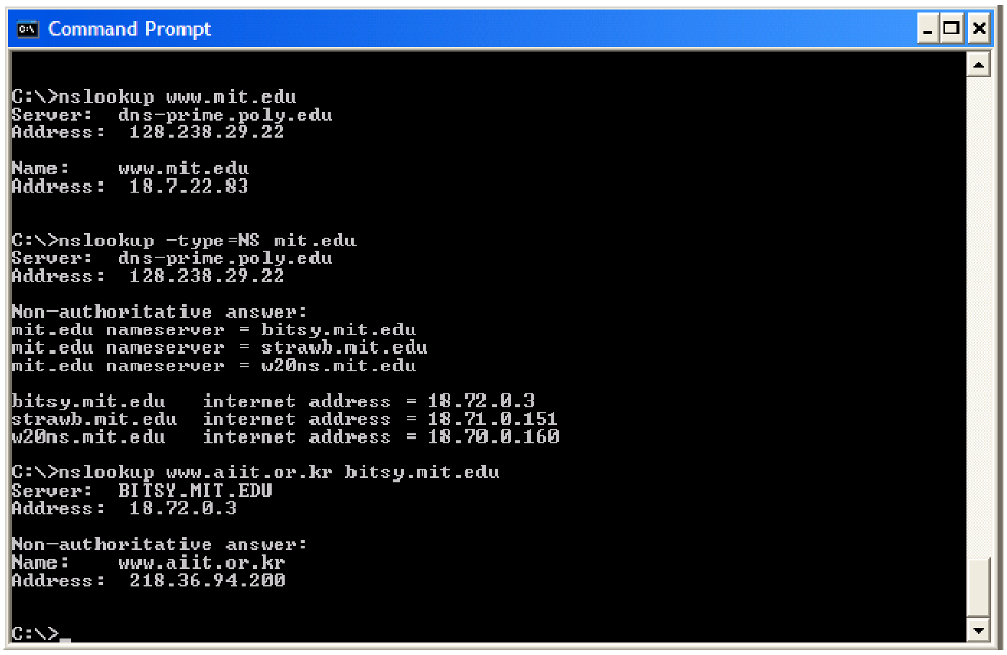
The below screenshot shows the results of three independent *nslookup* commands (displayed in the Windows Command Prompt). In this example, the client host is located on the campus of Polytechnic University in Brooklyn, where the default local DNS server is dns-prime.poly.edu. When running *nslookup*, if no DNS server is specified, then *nslookup* sends the query to the default DNS server, which in this case is dnsprime. poly.edu. Consider the first command:

```
nslookup www.mit.edu
```

In words, this command is saying "Please send me the IP address for the host www.mit.edu." As shown in the screenshot, the response from this command provides two pieces of information: (1) the name and IP address of the DNS server that provides the answer; and (2) the answer itself, which is the host name and IP address of www.mit.edu. Although the response came from the local DNS server at Polytechnic University, it is quite possible that this local DNS server iteratively contacted several other DNS servers to get the answer, as described in Section 2.5 of the textbook.
Now consider the second command:

```
nslookup www.aiit.or.kr bitsy.mit.edu
```

In this example, we indicate that we want to the query sent to the DNS server bitsy.mit.edu rather than to the default DNS server (dns-prime.poly.edu). Thus, the query and reply transaction takes place directly between our querying host and bitsy.mit.edu. In this example, the DNS server bitsy.mit.edu provides the IP address of the host www.aiit.or.kr, which is a web server at the Advanced Institute of Information Technology (in Korea).



```
C:\>nslookup www.mit.edu
Server:  dns-prime.poly.edu
Address:  128.238.29.22

Name:    www.mit.edu
Address:  18.7.22.83


C:\>nslookup -type=NS mit.edu
Server:  dns-prime.poly.edu
Address:  128.238.29.22

Non-authoritative answer:
mit.edu nameserver = bitsy.mit.edu
mit.edu nameserver = strawb.mit.edu
mit.edu nameserver = w20ns.mit.edu

bitsy.mit.edu   internet address = 18.72.0.3
strawb.mit.edu  internet address = 18.71.0.151
w20ns.mit.edu   internet address = 18.70.0.160

C:\>nslookup www.aiit.or.kr bitsy.mit.edu
Server:  BITSY.MIT.EDU
Address:  18.72.0.3

Non-authoritative answer:
Name:    www.aiit.or.kr
Address:  218.36.94.200


C:\>_
```

Now that we have gone through a few illustrative examples, you are perhaps wondering about the general syntax of *nslookup* commands. The syntax is:

```
nslookup –option1 –option2 host-to-find dns-server
```

In general, *nslookup* can be run with zero, one, two or more options. And as we have seen in the above examples, the dns-server is optional as well; if it is not supplied, the query is sent to the default local DNS server.
Now that we have provided an overview of *nslookup*, it is time for you to test drive it yourself. Do the following (and write down the results):

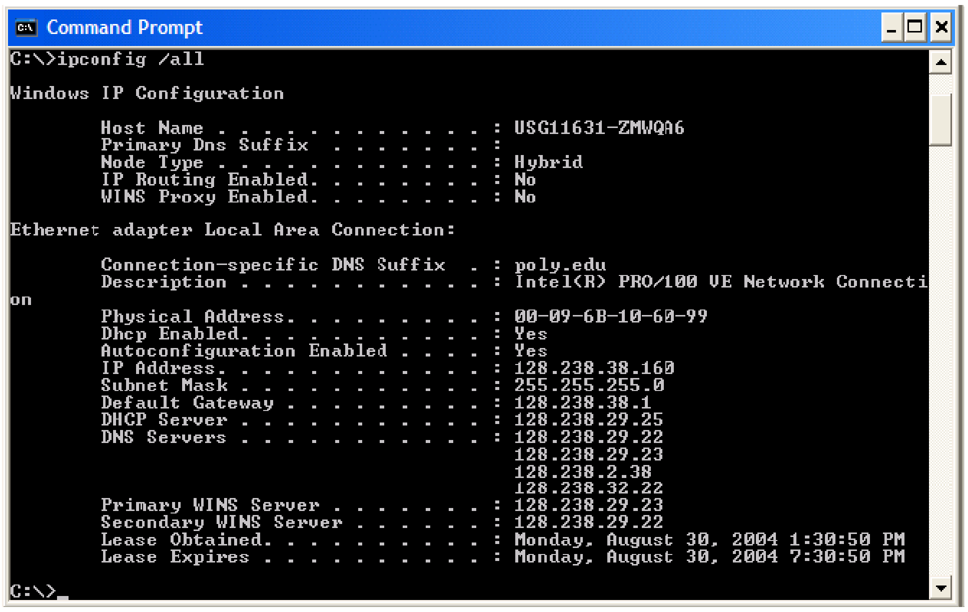
1. Run *nslookup* to obtain the IP address of a Web server in Asia.
2. Run *nslookup* so that one of the DNS servers obtained in Question 2 is queried for the mail servers for Yahoo! mail.

<u>ipconfig</u>

ipconfig (for Windows) and *ifconfig* (for Linux/Unix) are among the most useful little utilities in your host, especially for debugging network issues. Here we'll only describe *ipconfig*, although the Linux/Unix *ifconfig* is very similar. *ipconfig* can be used to show your current TCP/IP information, including your address, DNS server addresses, adapter type and so on. For example, if you want to see all this information about your host, simply enter:

```
ipconfig \all
```

into the Command Prompt, as shown in the following screenshot.



```
C:\>ipconfig /all

Windows IP Configuration

        Host Name . . . . . . . . . . . . : USG11631-ZMWQA6
        Primary Dns Suffix  . . . . . . . :
        Node Type . . . . . . . . . . . . : Hybrid
        IP Routing Enabled. . . . . . . . : No
        WINS Proxy Enabled. . . . . . . . : No

Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . : poly.edu
        Description . . . . . . . . . . . : Intel(R) PRO/100 VE Network Connecti
on
        Physical Address. . . . . . . . . : 00-09-6B-10-60-99
        Dhcp Enabled. . . . . . . . . . . : Yes
        Autoconfiguration Enabled . . . . : Yes
        IP Address. . . . . . . . . . . . : 128.238.38.160
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 128.238.38.1
        DHCP Server . . . . . . . . . . . : 128.238.29.25
        DNS Servers . . . . . . . . . . . : 128.238.29.22
                                            128.238.29.23
                                            128.238.2.38
                                            128.238.32.22
        Primary WINS Server . . . . . . . : 128.238.29.23
        Secondary WINS Server . . . . . . : 128.238.29.22
        Lease Obtained. . . . . . . . . . : Monday, August 30, 2004 1:30:50 PM
        Lease Expires . . . . . . . . . . : Monday, August 30, 2004 7:30:50 PM

C:\>_
```

ipconfig is also very useful for managing the DNS information stored in your host. In Section 2.5 we learned that a host can cache DNS records it recently obtained. To see these cached records, after the prompt C:\> provide the following command:

```
ipconfig /displaydns
```

Each entry shows the remaining Time to Live (TTL) in seconds. To clear the cache, enter

```
ipconfig /flushdns
```

Flushing the DNS cache clears all entries and reloads the entries from the hosts file.

Tracing DNS with Wireshark

Now that we are familiar with *nslookup* and *ipconfig*, we're ready to get down to some serious business. Let's first capture the DNS packets that are generated by ordinary Websurfing activity.

- Use *ipconfig* to empty the DNS cache in your host.
- Open your browser and empty your browser cache. (With Internet Explorer, go to Tools menu and select Internet Options; then in the General tab select Delete Files.)
- Open Wireshark and enter "ip.addr == your_IP_address" into the filter, where you obtain your_IP_address (the IP address for the computer on which you are running Wireshark) with *ipconfig*. This filter removes all packets that neither originate nor are destined to your host.
- Start packet capture in Wireshark.
- With your browser, visit the Web page: http://www.ietf.org
- Stop packet capture.

Answer the following questions:

3. Locate the DNS query and response messages. Are they sent over UDP or TCP?

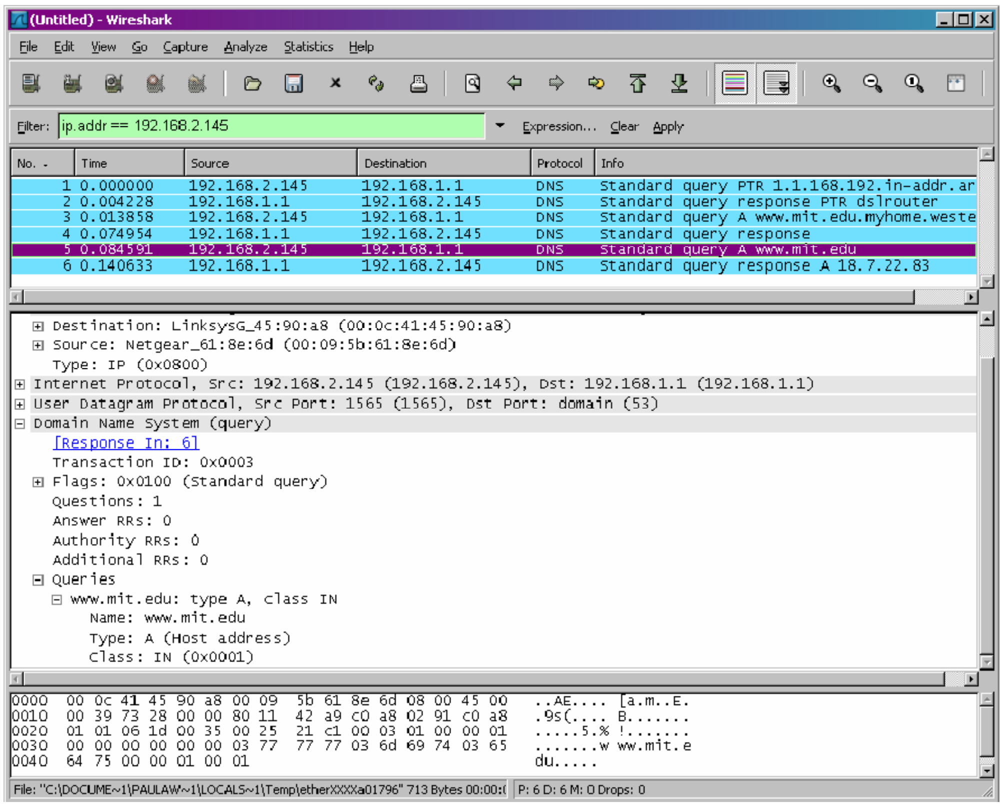
4. What is the destination port for the DNS query message? What is the source port of DNS response message?
5. To what IP address is the DNS query message sent? Use *ipconfig* to determine the IP address of your local DNS server. Are these two IP addresses the same?
6. Examine the DNS query message. What "Type" of DNS query is it? Does the query message contain any "answers"?
7. Examine the DNS response message. How many "answers" are provided? What does each of these answers contain?
8. Consider the subsequent TCP SYN packet sent by your host. Does the destination IP address of the SYN packet correspond to any of the IP addresses provided in the DNS response message?
9. This web page contains images. Before retrieving each image, does your host issue new DNS queries?

Now let's play with *nslookup*2.

- Start packet capture.
- Do an *nslookup* on www.mit.edu
- Stop packet capture.

You should get a trace that looks something like the figure below.

We see from the above screenshot that *nslookup* actually sent three DNS queries and received three DNS responses. For the purpose of this assignment, in answering the following questions, ignore the first two sets of queries/responses, as they are specific to *nslookup* and are not normally generated by standard Internet applications. You should instead focus on the last query and response messages.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: ip.addr == 192.168.2.145
Expression... Clear Apply

```
No. -  Time      Source         Destination    Protocol  Info
    1 0.000000   192.168.2.145  192.168.1.1    DNS       Standard query PTR 1.1.168.192.in-addr.ar
    2 0.004228   192.168.1.1    192.168.2.145  DNS       Standard query response PTR dslrouter
    3 0.013858   192.168.2.145  192.168.1.1    DNS       Standard query A www.mit.edu.myhome.weste
    4 0.074954   192.168.1.1    192.168.2.145  DNS       Standard query response
    5 0.084591   192.168.2.145  192.168.1.1    DNS       Standard query A www.mit.edu
    6 0.140633   192.168.1.1    192.168.2.145  DNS       Standard query response A 18.7.22.83
```

```
  ⊞ Destination: LinksysG_45:90:a8 (00:0c:41:45:90:a8)
  ⊞ Source: Netgear_61:8e:6d (00:09:5b:61:8e:6d)
    Type: IP (0x0800)
⊞ Internet Protocol, Src: 192.168.2.145 (192.168.2.145), Dst: 192.168.1.1 (192.168.1.1)
⊞ User Datagram Protocol, Src Port: 1565 (1565), Dst Port: domain (53)
⊟ Domain Name System (query)
    [Response In: 6]
    Transaction ID: 0x0003
  ⊞ Flags: 0x0100 (Standard query)
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  ⊟ Queries
    ⊟ www.mit.edu: type A, class IN
        Name: www.mit.edu
        Type: A (Host address)
        Class: IN (0x0001)
```

```
0000  00 0c 41 45 90 a8 00 09  5b 61 8e 6d 08 00 45 00   ..AE.... [a.m..E.
0010  00 39 73 28 00 00 80 11  42 a9 c0 a8 02 91 c0 a8   .9s(.... B.......
0020  01 01 06 1d 00 35 00 25  21 c1 00 03 01 00 00 01   .....5.% !.......
0030  00 00 00 00 00 00 03 77  77 77 03 6d 69 74 03 65   .......w ww.mit.e
0040  64 75 00 00 01 00 01                               du.....
```

File: "C:\DOCUME~1\PAULAW~1\LOCALS~1\Temp\etherXXXXa01796" 713 Bytes 00:00:(P: 6 D: 6 M: 0 Drops: 0

10. What is the destination port for the DNS query message? What is the source port of DNS response message?
11. To what IP address is the DNS query message sent? Is this the IP address of your default local DNS server?
12. Examine the DNS query message. What “Type” of DNS query is it? Does the query message contain any “answers”?
13. Examine the DNS response message. How many “answers” are provided? What does each of these answers contain?
14. Provide a screenshot.