New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Course Title: Data Communication & Networks
Course Number: CSCI-GA.2662-001
Instructor: Jean-Claude Franchitti
Session: 3

Lab #3 – Application Layer

I. Due

Thursday October 7, 2021 at the beginning of class.

II. Objectives

1. Understand the basic principles of computer networks and related protocols
2. Understand how to analyze application layer protocols
3. Develop a simple network application, explore and understand DNS protocol.

III. References

1. Slides and handouts posted on the course Web site
2. Textbook chapters as applicable

IV. Software Required

1. Microsoft Word.
2. Win Zip as necessary.
3. Wireshark,
4. Docker.
5. IBM Cloud
6. Kubernetes (K8s).

V. Lab Instructions

1. Problem 1 – Protocols Analysis:

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

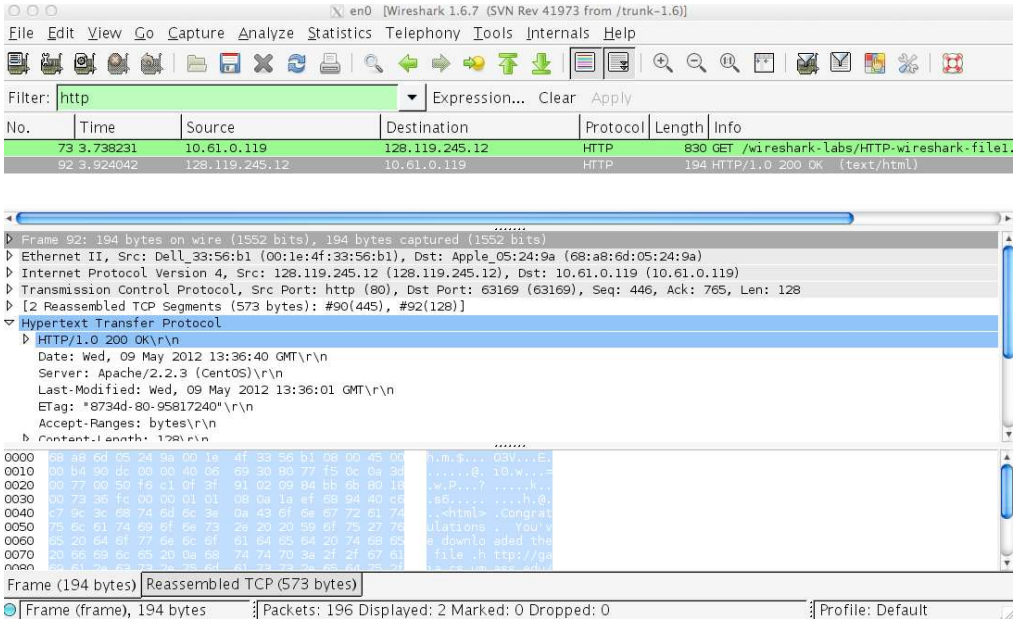
security. Before beginning these labs, you might want to review Section 2.2 of the text.

a. 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. If you are unable to run Wireshark on a live network connection, you can



download a packet trace that was created when the steps above were followed.[1]

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 is 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, un-displayed 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. When answering the following questions, you should print out the GET and response messages (see the introductory Wireshark lab for an explanation of how to do this) and indicate where in the message you've found the information that answers the following questions. When you hand in your assignment, annotate the output so that it's clear where in the output you're getting the information for your answer (e.g., for our classes, we ask that students markup paper copies with a pen, or annotate electronic copies with text in a colored font).

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

---

[1] Download the zip file http://gaia.cs.umass.edu/wireshark-labs/wireshark-traces.zip and extract the file http-ethereal-trace-1. The traces in this zip file were collected by Wireshark running on one of the author's computers, while performing the steps indicated in the Wireshark lab. Once you have downloaded the trace, you can load it into Wireshark and view the trace using the *File* pull down menu, choosing *Open*, and then selecting the http-ethereal-trace-1 trace file. The resulting display should look similar to Figure 1. (The Wireshark user interface displays just a bit differently on different operating systems, and in different versions of Wireshark).

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.

b. The HTTP conditional GET/response interaction

Recall from Section 2.2.5 of the textbook, 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 Firefox, select *Tools->Clear Recent History* and check the Cache box, 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.
- (*Note:* If you are unable to run Wireshark on a live network connection, you can use the http-ethereal-trace-2 packet trace to answer the questions below; see footnote 1. This trace file was gathered while performing the steps above on one of the author's computers.)

Answer the following questions:

1. 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?
2. Inspect the contents of the server response. Did the server explicitly return the contents of the file? How can you tell?
3. 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?
4. 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.

c. 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.
- (*Note:* If you are unable to run Wireshark on a live network connection, you can use the http-ethereal-trace-3 packet trace to answer the questions below; see footnote 1. This trace file was gathered while performing the steps above on one of the author's computers.)

In the packet-listing window, you should see your HTTP GET message, followed by a multiple-packet TCP 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 textbook) 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 45400 bytes is too large to fit into 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. Earlier versions of Wireshark used the "Continuation" phrase to indicated that the entire content of an HTTP message was broken across multiple TCP segments.. We stress here that there is no "Continuation" message in HTTP!

Answer the following questions:

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

Note: In some versions of Wireshark, instead of separate packets for each segment, Wireshark shows one packet. If you look at the part above Hypertext Transfer Protocol, you should see a section titled: "… Reassembled TCP Segments…". Please answer the questions according to your version.

d. 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 gaia.cs.umass.edu web site. The image of the cover for our 5th edition (one of our favorite covers) is stored at the caite.cs.umass.edu server. (These are two different web servers inside cs.umass.edu).
- Stop Wireshark packet capture and enter "http" in the display-filter-

specification window, so that only captured HTTP messages will be displayed.

- (*Note:* If you are unable to run Wireshark on a live network connection, you can use the http-ethereal-trace-4 packet trace to answer the questions below; see footnote 1. This trace file was gathered while performing the steps above on one of the author's computers.)

Answer the following questions:

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

e. 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.
- (*Note:* If you are unable to run Wireshark on a live network connection, you can use the http-ethereal-trace-5 packet trace to answer the questions below; see footnote 2. This trace file was gathered while performing the steps above on one of the author's computers.)

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:

1. What is the server's response (status code and phrase) in response to the initial HTTP GET message from your browser?
2. 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:

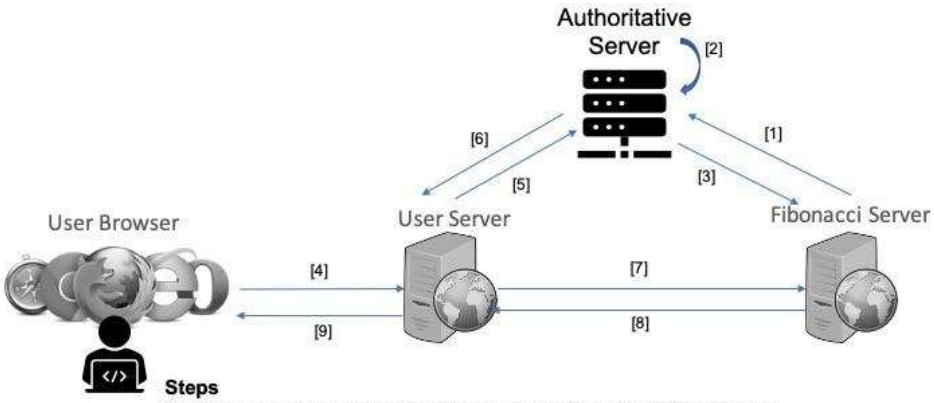
(d2lyZXNoYXJrLXN0dWRlbnRzOm5ldHdvcms=)

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.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 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 of the textbook, there are ways to make WWW access more secure. However, we'll clearly need something that goes beyond the basic HTTP authentication framework!

2. Problem 2 – Simplified authoritative server for a network of applications:

Figure below shows an overview of the system that you need to develop.



Steps
[1] Fibonacci Server registers its hostname with Authoritative Server
[2] Authoritative Server creates a DNS record for the Fibonacci server.
[3] Authoritative Server a response back indicating the success or failure.
[4] User visit:
http://IP_HTTP_SERVER:PORT/fibonacci?hostname=fibonacci.com&number=10&as_ip=3.4.5.6
[5] User server parse the hostname from the query and query the DNS Authoritative server via DNS query
[6] Authoritative Server returns back the IP address of the Fibonacci HTTP Server
[7] User server request http://FIBONACCI_SERVER_IP/fibonacci?number=8
[8] Fibonacci Server returns back the answer with 200 code.
[9] User server returns the result to the user

The system that you will implement has 3 components:

a. **User server (US)** is a simple HTTP web server (e.g. Flask), running in port **8080**, that accepts a GET HTTP requests in path:

   "/fibonacci?hostname=fibonacci.com&fs_port=K&number=X&as_ip= Y&as_port= Z"

   The path accepts five parameters: **hostname and fs_port** which contains the hostname and port number of the server which will be queried to get a response for Fibonacci number for a given **sequence number X**. And **as_ip, as_port** which are the IP address and port number of the Authoritative Server (AS). If any of the parameters are missing, the server should return HTTP code **400** indicating a bad request. If the request is successful, it should return HTTP code **200** with the Fibonacci number for the sequence number X. the only problem is that US does not know the IP address of the given hostname and therefore needs to query its authoritative DNS server to learn about it.

b. **Fibonacci Server (FS)** is an HTTP web server, running in port **9090**, that provides the Fibonacci value for a given sequence number X. In order to achieve this objective, FS performs the following:

   1. **Hostname Specification**: FS accepts a HTTP PUT request at path "/**register**" where the body contains the name and IP address of the server (FS) and IP address of the Authoritative Server (AS). You can

choose any name of your choice, let's say "fibonacci.com". The body should contain **a json object** as below. FS should parse the body and noted that hostname and IP and IP, port of the AS and moved to step 2 for registration to authoritative server.

```
{
    "hostname":
    "fibonacci.com",
    "ip": "172.18.0.2",
    "as_ip": "10.9.10.2",
    "as_port": "30001"
}
```

2. **Registration to Authoritative Server** Once the request is retrieved at path "/register", the hostname needs be registered with Authoritative server (AS) via UDP on port **53533**. Your DNS message should include (Name, Value, Type, TTL) where the type in this case is "A" and TTL in **seconds**. Below is a simplified DNS registration request (please follow the format as below -- without the dashes--, each line ends with a new line):

   TYPE=A
   NAME=fi
   bonacci.co
   m
   VALUE=I
   P_ADDR
   ESS
   TTL=10
   ----

3. Once registration is successful, returns back a HTTP response with code **201**.

4. Serve a path in "**/fibonacci?number=X**" which accepts HTTP GET request and returns Fibonacci number for the sequence number X. Return **200** as a status code if successful. If X is not an integer (for example a string) return **400** indicating the bad format.

c. Authoritative Server (AS) is the authoritative server for US. It has two duties. First is to handle the registration requests to pair hostnames to IP, second is to be able to respond to DNS queries from clients.

   i. **Registration**: Accept a UDP connection on port 53533 and register Type A DNS record into the database. Hint: You need to store the value in somewhere persistent, for example in a file

so that you can later respond to DNS queries.

ii. **DNS Query**: Respond to DNS Query on port 53533. Query should include: (Name, Type). If the message conforms to this, server will return the IP address in a DNS message of form: (Name, Value, Type, TTL), retrieved from the file (note that AS can distinguish registration requests from DNS queries by simply looking at the fields provided). Here is a sample request and response:

Request:

TYPE=A
NAME=fibonacci.com

Response:

TYPE=A
NAME=fib
onacci.com
VALUE=IP
_ADDRES
S TTL=10

Problem 2 will be graded based on the following. Please follow the instructions carefully for including the port numbers and message formats. Any deviation from the specification above will result losing points in the assignment:

a. US responds to requests in path /fibonacci as specified above.
b. FS accepts a registration request in /register as specified above.
c. FS responds to GET requests in /fibonacci.
d. AS performs a registration requests as specified above.
e. AS provides DNS record for a given query as specified above.

Please submit the following:

a. Create **dns_app** folder in Github and create three separate folders within it: "**US**", "**FS**" and "**AS**".
b. For each folder provide **Dockerfile** and all the files that Dockerfile needs in order to run the specific server.
c. Finally make the zip of "**dns_app**" folder and upload to NYU Classes.
d. Your final commit date to the folder in Github will be used to grade the submission.

Extra Credit: Run your application in IBM Cloud K8s cluster. Provide a single file (containing your deployment and service) that can be used to deploy your whole application with: "kubectl apply -f deploy_dns.yml". Be

sure to expose your ports with nodePort. Name the single file as "**deploy_dns.yml"**. Also note that K8s does not allow nodePort except for the range of 30000- 32767. Therefore, make sure that in your .yaml file, specify AS nodePort number as **30001**, FS port number as **30002** and US port number as **30003**.

You can take a look at this link to learn more about how to create this deployment file:

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a- deployment

Tips:

To build an image from Dockerfile, use the following command (change the tag accordingly and notice the . at the end which specifies the context of the build):

```
docker build -t bulutmf/fs:latest .
```

To get servers running within Docker containers communicate with each other, you can create a Docker network with the following command:

```
docker network create N_NAME
```

Once created run your containers by specifying the network name with the following command (change the parameters accordingly):

```
docker run --network N_NAME --name C_NAME -p 53533:53533/udp -it bulutmf/as:latest
```

Containers that are running within the same network should be able to communicate with each other. You can learn the IP address of your container by inspecting the network that you created with the following command:

```
docker inspect N_NAME
```

VI. Deliverables

1. Electronic:

   Your lab assignment files must be submitted via NYU Brightspace.

   For part V.1, your lab assignment file must be submitted via NYU Classes. Name the file "**firstname_lastname_lab_#.docx**" (e.g., "john_doe_lab_3.docx"). The

file must be created and sent by the beginning of class. After the class period, the homework is late. The email clock is the official clock.

For part V.2, submit all your code to NYU Classes (as a .zip file). Name the file "**firstname_lastname_lab_#.zip**" (e.g., "john_doe_lab_3.zip"). The file must be created and sent by the beginning of class. After the class period, the homework is late. The email clock is the official clock.

Also make sure to commit your programming question to Github. We will use the last commit time for grading.

- In Github create a folder for the programming exercise and name it "**dns_app**". Make sure to upload your code in there.

2. Cover page and other formatting requirements:

   The cover page supplied on the next page must be the first page of your lab assignment file.

   Fill in the blank area for each field.

   NOTE:

   The sequence of the electronic submission is:

   3. **Cover sheet**
   4. **Lab Assignment Answer Sheet(s)**

3. Grading guidelines:

   Assignment Layout (15%)
   - o Lab Assignment is neatly assembled on 8 1/2 by 11 layout.
   - o Cover page with your name (last name first followed by a comma then first name), username and section number with a signed statement of independent effort is included.
   - o File name is correct.

   Answers to Individual Questions (85%):
   - o Answers to all questions are complete and correct.
   - o Assumptions provided as required.

   (100 points total, all questions weighted equally)

VII. Sample Cover Sheet

Name _____ Date: _____
(last name, first name)
Section: _____

Lab #3 – Application Layer

Total in points (100 points total): _____

Professor's Comments:

Affirmation of my Independent Effort: _____
(Sign here)