CS 396: Mini Project 1

Due Date: 11:59pm Friday February 12

Through Moodle, turn in your four code files (i.e., there's two clients and two servers), screenshots of each app working properly, and screenshot of the Wireshark capture. Your code should be readable (i.e., Coding Style = 15 points).

1. **My First Web Server**

   Time to develop a web server capable of handling HTTP requests. The goal here is to get more familiar with creating sockets, binding sockets to a port, and the HTTP packet (e.g., creating a proper header format).

   Let's start by keeping it simple and creating a server that handles one request at a time. The web server should

   (a) accept an HTTP request,

   (b) parse the accepted HTTP request,

   (c) retrieve an object from the server's side,

   (d) if object exists, craft an HTTP response which includes the object in the data field,

   (e) if object does not exist, craft an appropriate "404" response.

   On Moodle you can find starter code for your server. Easy mode: Places for you to focus on are bookended with # Your code starts here and # Your code starts here. Hard mode: Start your own code from scratch. Using starter code (or not) does not deduct (or add) points.

   Web Server Testing

   Create an HTML file in the same directory your server is in. Run your server program. Find the IP address of the host running the server. From another host (e.g., ask a classmate), open a browser and provide the corresponding URL. Also, request an object that does not exist on the server.

   For example: If your HTML file is named "ilovenetworks.html", your IP address is "123.42.90.210" and your server is listening on port 6789, you would open your browser and type in the following (without quotes) to the address bar "http://123.42.90.210:6789/ilovenetworks.html". Not including the ":6789" will make the browser assume the default port 80.

   My First Web Client

   Create a super simple web client to go with your server. Your client should

   (a) take as input the server IP address, port at which the server listens, and object (with path) stored on the server

   (b) connect to the server using a TCP connection,

   (c) send an HTTP request with GET method to the server,

   (d) display server response message as output.

2. **Wireshark doo, doo, doo, doo, doo, doo**

   Use Wireshark to catch your web client and web server messages.

3. **You the ping now, doge!**

   Had fun running *traceroute*? Well let's make our own ping application on UDP to see some of those packets get lost as if into the vastness that is the internet . We will follow the client server model and use UDP as

our transport protocol of choice[1] to communicate between the two. The client will send a packet of data to a remote host (e.g., ask a classmate for help), the server will return the data back to the client (i.e., "echoing").

Err!

Assuming you'll be running this on Reed's network, which runs nicely, and building on HWK#1 let's introduce an error simulator. The error simulator should use random chance to decide if the server will or will not echo a ping packet. The starter code on Moodle for the server has bookending comments for where to include this.

Ping Client

Write the client side program for our ping application. Your client should

(a) send ten pings to the server,

(b) if the ping is echoed, print the response message from the server,

(c) if the ping is echoed, calculate and print the round trip time (RTT) for each packet,

(d) if the ping is not echoed, print "*" in lieu of the RTT,

(e) compute and display the average delay and packet loss rate (i.e., received / sent).

Pro tip: While developing your code, feel free to test on your own machine by running the server code first, and using your localhost IP (essentially make your computer ping itself). Once your code is "done", test it by having the server run on a remote machine (e.g., ask a classmate for help).

[1] Ping packets actually use Internet Control Message Protocol (ICMP, defined in RFC 792), but let's keep it simple.