CS168 Programming Assignment 4: Tethering

Assignment Out: April 23, 2012
Assignment Due: May 6, 2012 @ demo time

---

1 Introduction

In this final project, you will create a custom tethering solution, enabling you to provide connectivity to a laptop in-need-of-internet. On some mobile platforms, the distribution of tethering applications (applications that allow network requests from a laptop to be proxied through the 3G connection of the smartphone) is disallowed. Thus, we will not be developing a native (Andorid, iOS, etc.) solution. Instead, we will be developing a web-based solution, thereby bypassing the app approval process and enabling tethering via any modern smartphone.

2 Components

The project consists of three parts:

1. A websocket server running on the computer in need of internet, which establishes a connection between the laptop and the smartphone device. The laptop is configured to redirect all of its network traffic over the websocket connection, so that IP packets are sent to the smartphone device.

2. A websocket server running on a remote server, accessible by public IP. The remote server serves as a NAT, by reading all received IP packets and sending them out to the open internet as though the machine itself initiated the requests.

3. An HTML5 web app, which enables communication between the client and server machines by piping data between two websocket connections.

Imagine the laptop in need of internet attempts to connect to Google.com. We will configure the laptop so that this network request (as raw IP packets) will be sent to the smartphone via a websocket connection. The smartphone will then relay these packets to a remote server running somewhere in the cloud. The remote server will manipulate the IP packets, so that the source address matches the IP address of the remote server. It will then send these requests out to the internet as though it initiated the request itself. When the remote server receives a response, it will again manipulate the IP packets so that the destination address is changed back to the IP address of the laptop. Then, it will send the response packets back to the smartphone, which will relay them to the laptop. The laptop will then assume it has received a normal network response.

2.1 The Client

The client consists of two key parts. First, we need a way to establish a connection to the smartphone. Second, we need to redirect all network traffic through this connection.

2.1.1 The Websocket Server

The laptop must be running a websocket server, which allows the HTML5 web app to connect to the laptop. In order for the smartphone and laptop to communicate, they need to share a common wireless network. The easiest way to do this is to create an *ad-hoc network* on the lapop. You should find instructions online for how to create an ad-hoc network for your laptop operating system. For Mac OS X:

- Click on the AirPort icon in your menu bar. From the AirPort menu, select "Create Network..."
- Provide a name for the network, and optionally provide a password.

When the HTML5 web app is loaded on your smartphone device, it will attempt to connect to a known, fixed IP address. We will be using 169.254.134.89. Before we can run the websocket server which will bind to that address, we must assign that IP address to the Wifi device of the laptop. To do so, run the following command, substituting `en1` with your Wifi device name.

```
sudo /sbin/ifconfig en1 inet 169.254.134.89 netmask 255.255.0.0 alias
```

This command associates the IP address 169.254.134.89 with the laptop, so that packets destined to 169.254.134.89 will be passed up the network stack, rather than dropped.

Now that we have an ad-hoc network created, and a known, fixed IP address to listen on, let's write the websocket server. For this part of the assignment, you are free to use any language and any networking libraries you wish. We recommend using Tornado (a Python web framework) for four reasons: (1) The websocket specification is still young, and many modules that claim to be websocket-spec compliant may fail you. (2) We've tested an implementation using Tornado, found it easy to use, and encountered no major problems. (3) As you will see in the next section, if you are using Linux, it is *much, much easier* to get the client side working if your language has an `ioctl()` syscall available. Since Python has this readily available, it is a nice choice of language. (4) On the server-side, you will need to do IP packet sniffing and manipulation. Python has a great library available for this. Writing both the client and server in Python will make your life easier.

Create a server that binds to IP address 169.254.134.89 on port 6354 (these are arbitrarily chosen, but known and fixed constants, which will be useful when implementing the web app). After binding, the server should listen for new websocket connections, then manage reading and writing to these websockets.

2.1.2 Tornado + Mobile Safari Compatibility

Planning to use Tornado? If you are using an iPhone as your mobile device, with Safari as the web browser, you will need to override `WebSocketHandler.allow_draft76()` to return `True`. Other-

wise, your websocket requests will never be accepted by the server.

2.1.3 The TUN Device

You need to have a way to get IP packets leaving the client to be sent through the websocket connection, rather than directly through the standard network interface. Rather than writing a kernel module to do this, there is a better way: to create a *virtual interface*. A virtual interface, also called a TUN/TAP device, looks to the IP stack just like a physical interface (e.g., `eth0`), but the packets sent to this interface are delivered *to a user-space program*. Likewise, when this user-space program writes packets to the virtual interface, they are delivered to the IP stack in the kernel, as if they had come from the network.

The difference between a TUN and a TAP interface is that the TUN interface works at the IP layer: it sees raw IP packets, while the TAP interface sees raw Ethernet frames. Since you are tunneling IP packets, you can use a TUN interface for this.

If you are using Mac OS X, you will need to download and install the TunTap package for Mac OS X[1]. Once installed, you can simply `os.open()` a file descriptor to `/dev/tun0` and start reading from and writing to the TUN device.

On Linux, things are a little more complicated[2]. Unlike on Mac OSX, on Linux you must open the file descriptor `/dev/net/tun`, and then issue a call to `ioctl()` with some parameters to initialize the device.

You should perform these TUN/TAP initializations (by opening the appropriate file descriptor, calling `ioctl()` if necessary) in your webserver implementation, so that the file descriptor is then available for reading and writing. This will allow your websocket server to write into the device (simulating that the laptop received a packet from the network) or read from the device (intercepting outgoing IP packets).

Once you have your TUN/TAP device set up, there's one final step. You need to instruct your laptop OS to take *all* outgoing network requests and send them into the TUN device instead of using the standard networking interface (i.e. `eth0`). To do so, you need to assign your TUN device a known IP address:

```
sudo ifconfig tun0 10.0.0.1 10.0.0.1 netmask 255.255.255.0 up
```

Then, modify the OS's IP routing table to delete the current default route, and route all network traffic through the tunnel:

```
sudo route delete default; sudo route add default 10.0.0.1
```

Since these commands must be issued *after* the TUN device is set up, we recommend issuing them from the webserver's initialization process, after the TUN config is finished.

Now, if you issue a network request from your laptop, you should be able to call `os.read()` on the file descriptor for the TUN device, and see the raw IP data flowing into your webserver

[1] `http://tuntaposx.sourceforge.net/download.xhtml`

[2] `http://backreference.org/2010/03/26/tuntap-interface-tutorial/`

process. This raw IP data needs to be sent to the smartphone via the websocket connection. We recommend using base64 encoding to encode the IP data over the websocket.

2.2 The Remote Server

2.2.1 Amazon EC2

You will be deploying your server implementation to Amazon EC2. The server needs to be publicly addressable by public IP. After all, the smartphone needs to be able to connect to the server via its 3G connection. EC2 provides a free usage tier, whereby new users can rent an EC2 Micro Instance for 750 hours without being charged. Learning to configure and deploy servers in the cloud is also a very useful skill that may prove helpful when you least expect it. If you already have your own server or would prefer not to use EC2 in favor of a different service, that's fine. However, the TA staff will only be assisting with EC2 configuration, and you will need to have root access on the server you do choose to use. To set up an EC2 instance:

1. Sign up for Amazon Web Services.
2. Open the AWS Management Console.
3. Click on the "EC2" tab.
4. Click "Launch Instance."
5. Click "Quick Launch Wizard."
6. Name the new instance (e.g. "TetherServer").
7. Create a new key pair (titled "tether" for instance) and download it to ∼/.ssh.
8. Select "Ubuntu Server Cloud Guest 11.10 (Oneiric Ocelot)" as the OS to use.
9. Click "Continue", then "Launch."

Now, the server is running in the cloud. But before you start configuring it further, we need to make sure a few ports are open. Otherwise, the server will just deny all incoming requests.

1. In the outline view on the left, click "Security Groups."
2. Select "quicklaunch-1" (the default security group used when launching instances via the Quick Launch Wizard).
3. At the bottom, you'll see an "Inbound" tab. Click it.
4. Select "HTTP" from the "Create a new rule:" dropdown. Make sure "Source:" is set to "0.0.0.0/0". Then click "Add Rule."

5. Select "Custom TCP rule" from the "Create a new rule:" dropdown. Make sure "Source:" is set to "0.0.0.0/0". Set "Port range:" to 8080. Then click "Add Rule."

6. Click "Apply Rule Changes."

Now, Amazon will allow incoming requests from anyone on port 80 (HTTP) and port 8080. We'll use port 8080 later for the websocket connection port.

With that, our server is set up! Time to ssh into it. You can find the IP address of your newly created EC2 instance under the EC2 dashboard by clicking "Running Instances."

1. chmod 600 ~/.ssh/tether.pem

2. ssh -i ~/.ssh/tether.pem ubuntu@[your-amazon-IP-address].amazonaws.com

Now you can go about installing various modules (`apt-get` and `easy_install` are your friends).

2.2.2 Websocket Server and NAT

You will need to write another websocket server to run on the remote server. It should listen for incoming websocket connections on port 8080 and receive base64 encoded IP data from the phone. Now, the remote server needs to issue a network request with the same IP data as received, except the source address field should be replaced with the IP address of the EC2 instance. You can find the IP address of the `eth0` device by using `ifconfig`.

If you are using Python, you may utilize Scapy[3], a powerful packet manipulation library. You should be able to decode the base64 encoded string and construct an IP packet out of the data. Then, replace the source IP address with the IP address of the `eth0` device. Finally, make sure you update the checksum to reflect the modified header. Now that the packet has been modified, it can be sent out to `eth0` as a regular network request.

In order to grab the response packets, you'll need to packet sniff `eth0`. Scapy has functions that allow you to do so. But, you don't want to send *all* incoming packets back to the smartphone! After all, the EC2 instance may be making lots of outgoing requests that are unrelated to your tethering. You'll need to come up with a way to only send back the relevant response packets to the phone, after you've sniffed them off the wire.

Note: Before executing Scapy scripts, it is necessary to disable the Linux Kernel's own responses. If the Linux Kernel is allowed to answer arriving network packets, it will answer with RST-ACK during the TCP handshake, because the Linux Kernel believes that no port is open. Scapy operates besides the Kernel (Paracommunication), therefore the Kernel has no knowledge of what Scapy is doing.

For TCP, disable the kernel's response with:

```
sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

[3] `http://www.secdev.org/projects/scapy/`

2.3 Static File Server

Finally, the remote server will need to serve the files needed to load the web app on the smartphone. Tornado has the functionality to do this, or you can even write a separate static file server (using Nginx, Apache, Node.js, etc.) if you wish.

2.4 The Web App

The web app should be pretty straightforward. It should open two Websocket connections:

```
var remote = "ws://[your-amazon-IP-address].amazonaws.com:8080/websocket/";
var local = "ws://169.254.134.89:6354/websocket";
```

Then, when it receives a message on the local websocket, it should write the message into the remote connection. Likewise, when it receives a message on the remote websocket, it should write it into the local connection.

We recommend having some timers to retry establishing the websocket connection if it fails the first time around. Adding some UI to the HTML page to indicate whether or not the connections have been established may be nice for debugging, though we don't require it. Likewise, adding detection for non-javascript-enabled browsers, non-websocket-enabled browsers, etc. are nice features, but not required.

Make sure you set the "binaryType" of the websockets to "arraybuffer", so that they are handling raw bytes.

The web app should be the easiest aspect of the project, so if you feel like you're doing a lot of work here, please consult with a TA.

In order to establish the connection to the laptop, make sure you join the ad-hoc network you created earlier on your smartphone. Then, a request to connect to 169.254.134.89:6354 will be handled by the ad-hoc network.

3 Recommended Strategy

We recommend you build this project in stages. First, set up your EC2 instance. You may run into problems configuring your EC2 instance and it's better to resolve those up front. The TAs are available to help you configure things if you get stuck. You'll also need your EC2 instance for the next part of the project, because you'll need a way to actually serve your web app (i.e. you'll need to be able to point your phone to a URL where it can actually access the web app).

Second, establish a websocket connection between your web app and your laptop in need of internet. Make sure they are able to connect and exchange messages before you even try setting up the TUN device. It'll be much easier to debug the connection when the nodes are sending "Hello World" messages than when they are trafficking all IP requests coming from your laptop.

Third, make sure your web app can connect to your EC2 instance. Again, just make sure that simple messages can be exchanged between your phone and the remote server. Once that's working, you can test the full circuit by sending a "Hello World" message from the laptop, ensure

it is received by the remote server, send a response from the remote server, and make sure it is received at the laptop. Once you have all of this working, you can move on to the TUN device configuration and NAT implementation, which are the difficult parts of the project.

4 Grading

You must hand in all of the source code that you use for the client, server and web app portions.

There will be an interactive demo/grading session on May 6th. You will demonstrate your tethering solution to a TA, then hand in your source code for evaluation. We will require you to open a few websites on your laptop in need of internet to demonstrate your solution. You should try to optimize your code so that page loads are nearly as fast as your 3G connection allows.

5 Limitations

Neither the remote server nor the client websocket server need to handle more than one websocket connection at a time. Imagine you are building this for yourself to tether data, and you'll only ever be connecting with a single smartphone.

While theoretically the architecture we've presented here could be used to tether *any* IP traffic, we are only requiring that it work for TCP and UDP. Further, you do not need to support any protocols other than IP (ping does not need to work, for instance). This should make the job of writing the NAT a little bit easier.

Additionally, we are not requiring you to setup DNS resolution on the client machine. This means that when you have your final solution, typing Google.com in your web browser is not required to work. But typing in http://173.194.43.4 should render the Google homepage.

Have fun!