CS 240 F17

Lab 1: Virtualization, sockets, RPCs

Hassan Alsibyani, Humam AlWassel, Guolei Sun, Marco Canini

Part 1

The first part is designed to get you acquainted with Amazon Web Services (AWS), Vagrant, and Docker tools if you haven't used them before. This is not meant to be an exhaustive guide to any of the tools and might not be completely self-contained.
For how to install Vagrant and Docker, please refer to the instructions on their respective websites:
https://www.vagrantup.com/
https://www.docker.com/

1.1 Create an AWS account

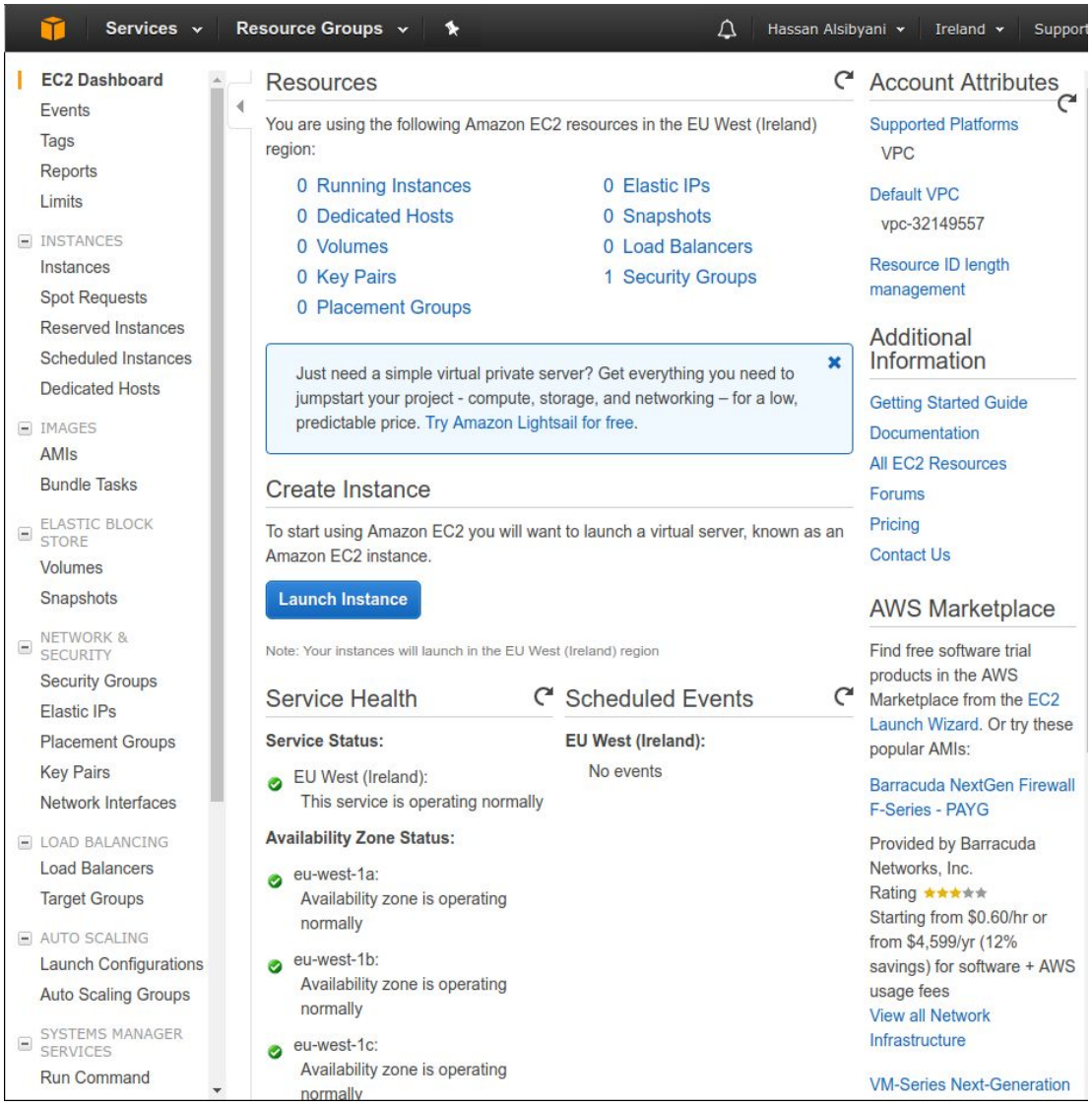
This one is straight forward. Go to www.awseducate.com/Application and create a student account. Make it as soon as possible if you haven't already since it requires to go through approval by AWS.

1.2 Run an instance on AWS

The VM service offering on AWS is called EC2 (Elastic Compute Cloud). You can see information regarding service pricing at: https://aws.amazon.com/ec2/pricing/.
For this lab, we suggest that you try to make use of Free Tier service by using the **t2.micro** instance type.

While you can use CLI tools or APIs to launch an instance, the browser GUI is gentler and might give you a better idea on the available options.

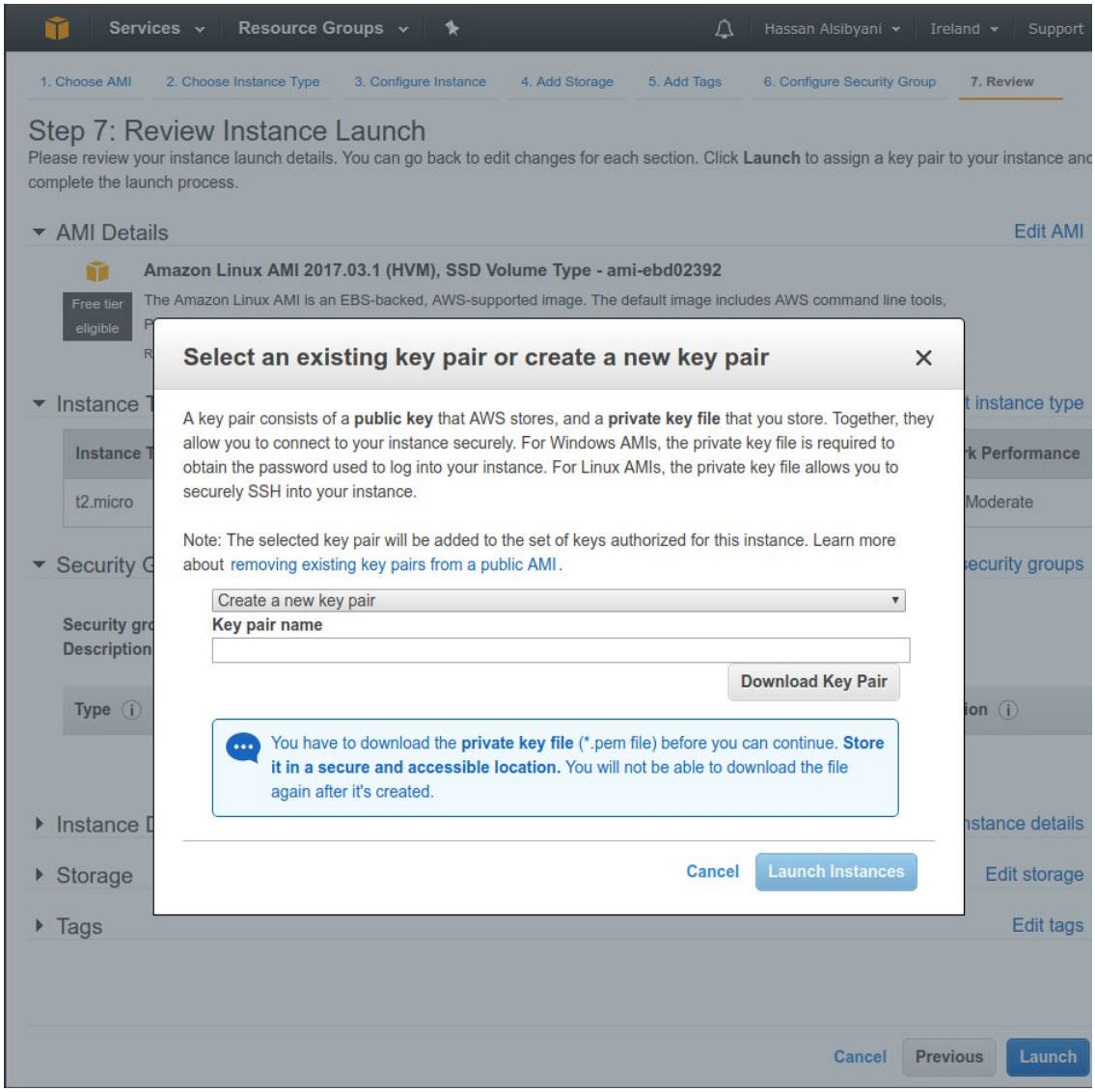
Login on AWS at https://console.aws.amazon.com. Go to the EC2 Dashboard and press the big blue **Launch Instance** button



From there you will see many options for different types of OSes, computation power, and network availability. You can choose the cheaper **t2.micro** instances since they are Free Tier eligible. As for the OS, Ubuntu 16.04 is a good choice.

By the end of the setup wizard, you will be asked to create a key pair. This key is essential to SSH into your instance. *Make sure you download the private key.*

If you are not familiar with key-based authentication for SSH, please consult this guide: https://www.digitalocean.com/community/tutorials/how-to-configure-ssh-key-based-authentication-on-a-linux-server. Please note that AWS generates the key for you.



Congrats, your instance is now getting set up and will launch soon. It should take just a few minutes to become ready. Go to your instances dashboard and you should see it. Right-click the instance and you will see a **connect** option, follow the instruction to SSH to your instance (hint: you will need the private key you just downloaded).

1.3 Provision a VM on your laptop with Vagrant

Vagrant is a great tool to provision VMs or containers to make them easier to share and reproduce especially in production environments. Their website provides clear explanations on what Vagrant is and how you can get started: https://www.vagrantup.com/intro/index.html. Make sure you install Vagrant before proceeding.

In this example, we are going to use VirtualBox as our hypervisor. We are going to use Ubuntu 16.04 as our guest OS. We will run an nginx web server inside the VM. We will need to configure Vagrant to forward TCP port 80 so we can access it on our host machine.

In order to use Vagrant, your settings/provisions needs to be defined in a Vagrantfile. Go somewhere in your working directory and run:

```
mkdir nginx_vagrant_example
cd nginx_vagrant_example
touch Vagrantfile
```

Now copy this in your Vagrantfile:

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.provision "shell", path: "reqs.sh"
  config.vm.network "forwarded_port", guest: 80, host: 8080, id: "nginx"
end
```

In general, it is important not to be a cargo cult developer and try to understand what is happening before we run it. This will use Ubuntu as the VM box, will provision the image by running reqs.sh and will forward port 80 from the running instance to 8080 on the localhost (this means that all the traffic sent to localhost, which is your machine, on port 8080 is automatically forwarded to the VM on its port 80). reqs.sh will be a simple file that installs and runs nginx. Create reqs.sh and paste the following in it:

```
apt-get -y update
apt-get -y install nginx
service nginx start
```

Now run `vagrant up --provider virtualbox` and Vagrant will install the Ubuntu image, launch it, and provision it. This will take some time. After it launches, you can go to http://localhost:8080/ or `curl localhost:8080` and you should see nginx running.
To get a shell inside the VM, you can run `vagrant ssh`.
When you are done, you can `vagrant halt` to stop the instance. To restart it, you can simply run `vagrant up` again (no need to provision it). If you run `vagrant destroy`, this command destroys the instance (and removes all of its state).

1.4 Create and run a container with Docker

Docker allows you to have some isolation with much lower overhead than a VM. Install and *run* the Docker daemon. Let us make another directory and create the Dockerfile

```
mkdir nginx_docker_example
```

```
cd nginx_docker_example
touch Dockerfile
```

Now paste this into the Dockerfile

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y nginx && service nginx start
EXPOSE 80
```

Now we will have to build our image by running `docker build -t="nginx_example" .` and now we can launch it by `docker run --name nginx-docker -p 8080:80 nginx_example` this does something similar to the Vagrant example above and you can go to localhost:8080 again and see the server running.
In fact, this was the long way of running an nginx server on Docker. Instead of creating an Ubuntu image and downloading nginx on top of it, you can directly use an pre-made nginx image and run it just by using

```
docker pull nginx
docker run --name nginx-dockerimg -p 8080:80 nginx
```

Docker provides you with many images and you can imagine how this can be useful when working on large projects on different machines that may contain different tools and environments.

1.5 Benchmark performance of nginx in a VM vs. a container

Extra task

This is an extra task. We recommend that you attempt this after you have completed all the other parts of this lab.

Do you wonder how much faster does ngnix run inside a container instead of when running in a VM?
You can try to find out!

To do so, try to reproduce the benchmarks from this blog post on the official ngnix website: https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/.
Please note that you will need to make some set up of your environment and install the `wrk` performance benchmarking tool in your host machine.
The setup details are described on this blog post: https://www.nginx.com/blog/nginx-plus-sizing-guide-how-we-tested/.

Part 2

Sockets are a representation of endpoints in networks that allows two different processes to communicate with each others, this could be in the same machine or in different machines. The use of sockets is common within many application, for example: HTTP, FTP, and IRC.

This second part is designed to let you practice the basics of socket programming. Our example is basic: a client will send a string (the request) to the server, and the server replies with a string (the response) that echoes the request. We provide a server program (server.go) that creates a TCP socket and listen on a port. When it receives a new connection, it spawns a go routine to handle the request. You will run this program in various environments and test it with the the nc tool. You will then write your own client program using socket programming in Go.

If you need some background on sockets or network programming in Go, please see: https://appliedgo.net/networking/.

2.1 Run a simple echo socket server program

Before this step, make sure you have a running installation of Go on your machine.
Place the provided program (`sever.go`) into your work directory and run it as:

```
go run server.go
```

You can see an output as "`Listening on localhost:3333`", which means that the program is running successfully and awaiting connections on port 3333. Let's test it.

2.2 Test the server

In this step, you need to test that the provided server program functions as expected.
On the same host as the server, run the following:

```
echo -n "test out the server" | nc localhost 3333
```

If all goes well, you should see this message:

```
Message received: test out the server
```

The server replied with our request, as expected.

The above works well because both the client (`nc`) and the server (`server.go`) are running on the same machine. What happens when we deploy our server in a different execution environment?
Let's try to run the program inside a VM on AWS as well as inside a Docker container.

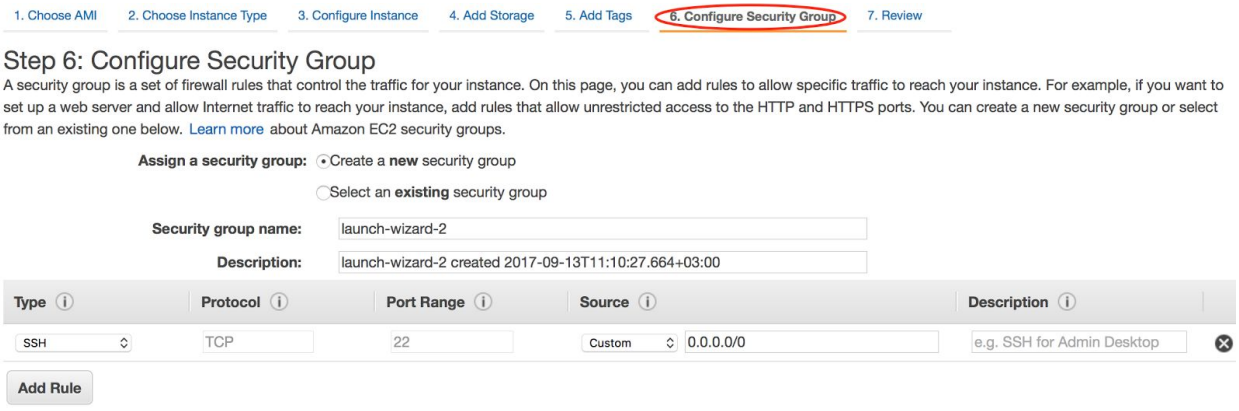
Now, create and launch an instance like we introduced in Part 1.
Once the instance is up, setup Go inside it. If you are using a Debian-based machine (like Ubuntu), type the following commands to setup Go.

```
sudo apt-get update
sudo apt-get install golang
```
Then run the server program and try to run the client on your machine/laptop.
Well, it seems that the server is not working anymore. Can you tell what is wrong?

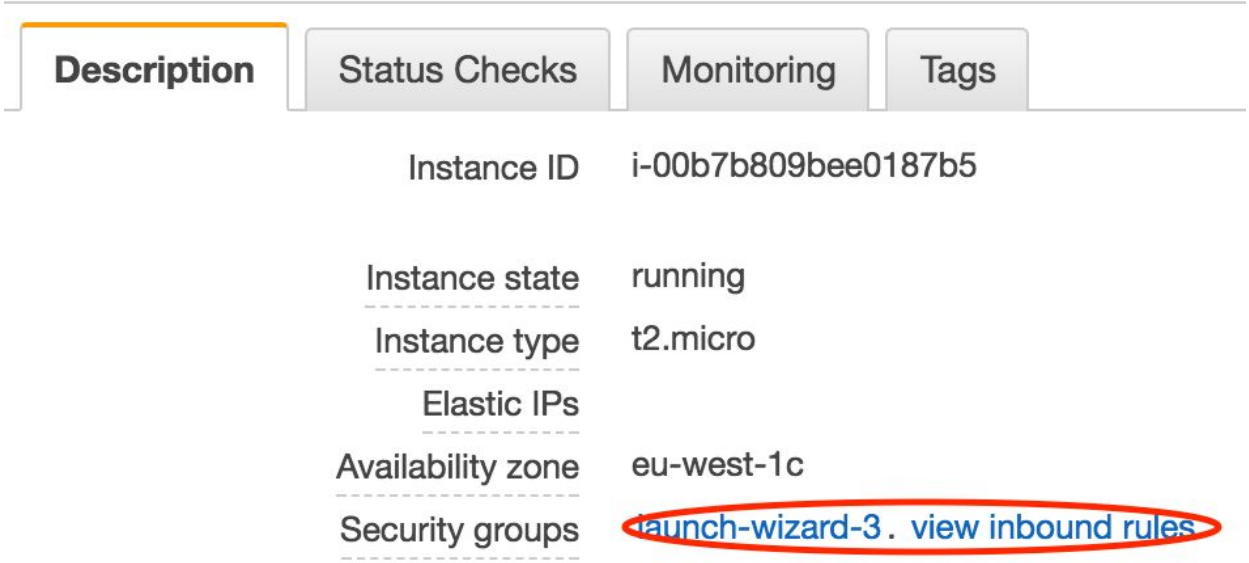
There are two issues that we need to solve.

First, when you run the server, have you notice that it outputs "`Listening on localhost:3333`". This means that the socket is bound to the localhost interface. This constraints the server to only accept connections if they are local. But we instead are trying to reach the VM via its public IP address.
To solve this issue, we need to change the interface that the socket is bound to.
Replace "`localhost`" with "`0.0.0.0`" in this line:
```
CONN_HOST = "localhost"
```

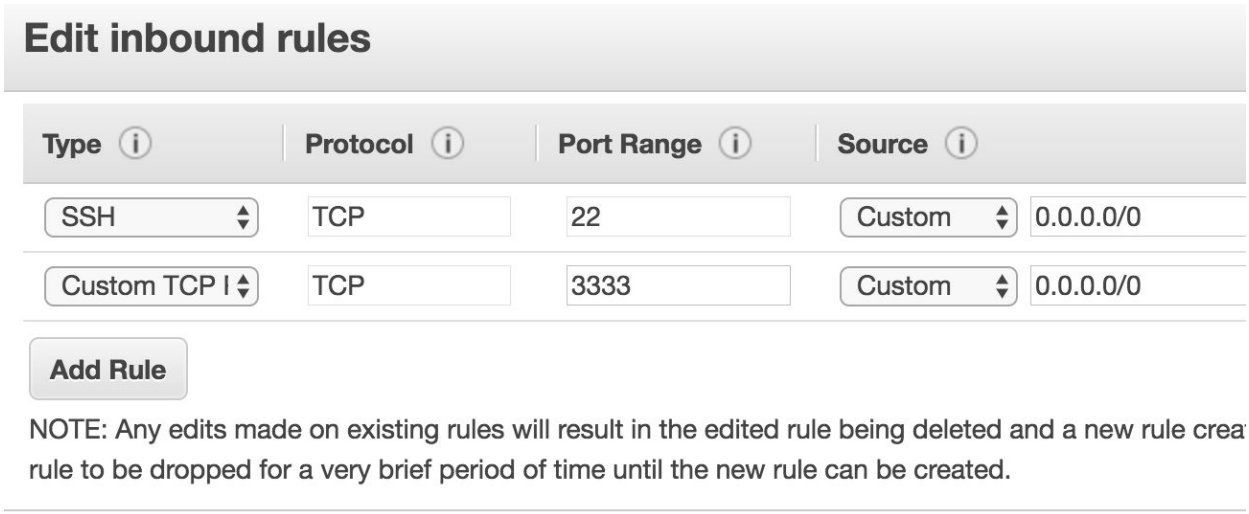
The second issue has to do with the security policy of the VM. By default, AWS applies a security group that only enables SSH connection via port 22.
If you recall, when you created the VM instance, the security group configuration looked something like this.



The good thing is that we can change this configuration at any time. Go to the EC2 Dashboard, select your VM and in the Description section click on the security group name. Below this is shown as "`launch-wizard-3`". Note that the name can be different in your setup.



Now let's open up port 3333. Click the **Add Rule** button to add another protocol as shown here.



At this point, you should be able to successfully use the server program running on the Cloud.

2.3 Write a socket client in Go and run it

Last but not least, so far you have just made use of the `nc` tool to connect and send a request to the server. It is now time to write your own socket client in Go.
Now write a socket client program. This client will:

1. connect to a specified server,
2. send a string of your choice to the server,
3. read the response from the server, and
4. output to stdout.

Basically it will do the same function for which we were using `nc` before.

The complete example code for client can be found on the course website. A few hints:

- Connect to the server
  `conn, _ := net.Dial("tcp", serverAddress + port)`
- Send a string to the server
  `fmt.Fprintf(conn, string)`
- Read the response
  `message, _ := bufio.NewReader(conn).ReadString('\n')`

2.4 Run the client in different environments

Extra task

This is an extra task. We recommend that you attempt this after you have completed all the other parts of this lab.

Once you have created your client program, try to run your client and the server in different environments (AWS, Vagrant VM, Docker container). See if there are combinations where the default network configuration requires tweaking for communication to happen (there are certainly some!).

Part 3

This part is designed to get you acquainted with Remote Procedure Calls (RPC). You will create a simple client and server RPC program using Go's `net/rpc` package.

3.1 What is RPC?

RPC is when a machine (client) calls a procedure/function that's executed on another machine (server) as if the client was just calling a local function that executes on its own machine. The server and client usually live on different machines that can reach each other by talking over network. An RPC goes through the following steps:

1. Client calls stub (local procedure call)
2. Client stub marshals parameters
3. Client OS sends message to server
4. Server OS passes message to server stub
5. Server stub unmarshals parameters
6. Server stub calls the server procedure
7. Trace back in reverse direction to return the results to the client

3.2 RPC in Go

You will use the `net/rpc` package to implement a simple client / server RPC program. `net/rpc` allows us to write RPC programs very easily in the following way:
On the server side:

- Create a server
  – Create a TCP server (or some other server to receive data)
  – Create a listener that will handle RPCs
  – Register the listener and accept inbound RPCs
- Write stub functions
  `func (t *T) MethodName(argType T1, replyType *T2) error`
- See https://golang.org/pkg/net/rpc/ for more details

On the client side :

- Create a client
  `client, err := rpc.DialHTTP("tcp", serverAddress + port)`
- Make an RPC
  ```
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  ```
- Unpack return values
  –Treat as any normal variable

For more background on RPC in Go, please see: https://talks.golang.org/2013/distsys.slide and http://blog.prevoty.com/writing-your-first-rpc-ingolang.

3.3 Create a simple client / server RPC program

- We are running a time server
- Goal is to implement an RPC client that uses Cristian's algorithm to get the server's clock time

  $$servertime = T_3 + \frac{(T_4 - T_1) + (T_3 - T_2)}{2}$$

  Where T_2 and T_3 are the timestamps returned from the server, T_1 and T_4 are the timestamps before and after the RPC call, respectively, on the client side. $T_4 - T_1$ is known as the round-trip delay time (RTT) between the client request and server response.
- Code skeleton for `sever.go` and `client.go` are available on the course website
- You will need need the `time` and `net/rpc` packages. *Beware the difference between Time and Duration!*
- You need to implement `GetServerTime` function with this signature:
  ```
  // TODO
  // GetServerTime implements Cristian's algorithm
  // 1. Keep track of the appropriate timestamps from the
  ```

```
//    local machine. Remember to get T1 close to the
//    beginning of the function!
// 2. Request T2 and T3 from the server via an RPC call
//    - serviceMethod: Listener.GetServerTimestamps
//    - args: Request
//    - reply: ServerTimestamps
// 3. Compute the server timestamp (watch out for
//    duration vs. time)
func GetServerTime(request *Request, client *rpc.Client) time.Time {}
```

- run your client like this

```
go run client.go [NetID] [ServerIP] [ServerPort]
```