Part 0.1

AWS, SSH Clients, Client Server Programming

This lab assumes that you have already created an account on AWS, Amazon's cloud computing platform. Everything we cover in this lab and the next uses the free tier of AWS, i.e. you won't need any credit in your account to use these services. However, you do have the option of claiming credits under AWS educate. You can do that simply by going to AWS Educate; joining as a student with your imperial email account and using a code provided to you. You can then sign back into your AWS account via AWS Educate and you should see credits in your account. However, in order to begin this lab, you can continue without these credits as well.

A detailed walk-through on this process, by Dr David Thomas, is available on blackboard. The walkthrough also takes you through the steps of installing an EC2 instance. However, all this and more is explained in detail in this manual.

1. Agenda of the lab

The goal of this lab is to learn how to:

- Set up and instantiate an EC2 instance under your AWS account
    - Paying attention to various security groups, making various selections of security groups and understanding the consequences of these selections.
- Communicating with the EC2 instance using SSH clients
    - Using PuTTY to install Python on your EC2 instance
    - Using FileZilla to transfer files – including, for example, python code – from the local machine to the server and vice versa
- TCP client and server
    - Writing and running a TCP server on the EC2 instance
    - Writing and running a TCP client on the local machine
    - Communicating between client and server

2. Lab Tasks

We will cover this agenda in a series of tasks described below.

2.1 Instantiating an EC2 instance

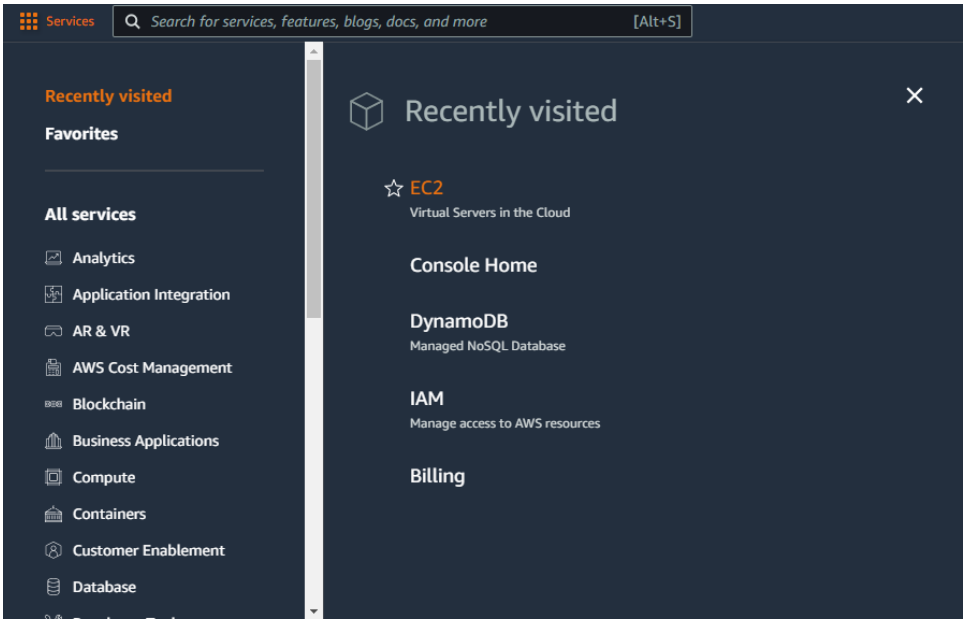
We begin by instantiating an EC2 instance.

Perform the following series of steps:

i. Sign-in to the AWS console as a *root* user.

The option of IAM (Identity and Access Manager) user is for the purpose of creating an entity (person or application) that can interact with AWS services under specified rights and permissions. We will use this option later, when using a database service in AWS, so it is useful to note its existence.

ii. **From the Services menu, select the EC2 option:**

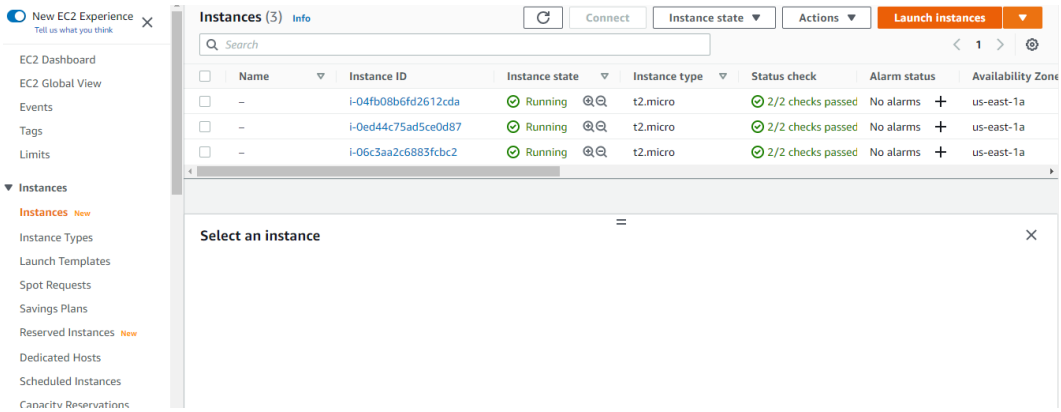


You can use the search bar on top to search for EC2 if it's not directly visible

EC2 stands for Elastic Compute Cloud. It allows you access to a virtual computer on the cloud. This is a full-fledged machine with an operating system of your choice and considerable processing power. You can run many useful applications, including your own programs, on your EC2 instance. For example, one of the programs we will be running on our EC2 instance is a TCP server written in Python. *Essentially the same TCP server we wrote in the Software Systems module.*

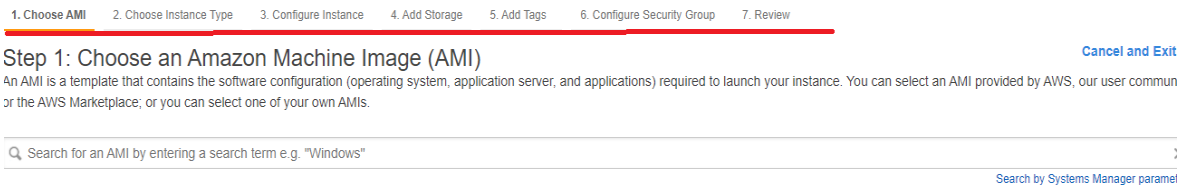
iii. Launch an EC2 instance.

Do this by clicking the Launch Instances button on the top right of the screen:



In the screen above, I already have three instances of EC2 running, however you will not see any running instances at the moment.

After clicking Launch Instances, we will be asked to specify various details of the instance. This process has 7 steps, under 7 tabs shown on the top of the screen. In the following I have underlined these tabs in red:

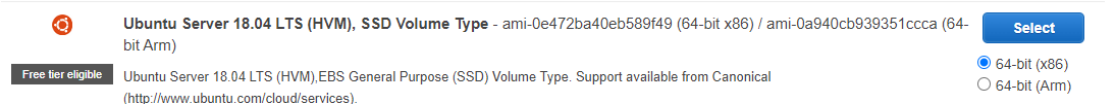


The steps are: Choose AMI, Choose Instance Type, Configure Instance, Add Storage, Add Tags, Configure Security Group, and Review. Following is a brief explanation of some of the more important steps:

1. Choose an Amazon Machine Image

This is where you specify the type of Processor, Storage and Operating System of your EC2 instance. As you'll see, some of the available options are eligible for the free tier. We'll be selecting one of these options, which should be sufficient for your projects; however, you can certainly explore more and make different selections to suit your purposes.

In the search bar, type in Ubuntu and select the following option:



The difference between the first two options is among the Ubuntu versions. For example, 20.04 runs a different version of the Linux Kernel. However, for our purposes it wouldn't make a real difference if you choose 20.04 LTS or 18.04 LTS. Here is a table of comparison between the versions: *https://computingforgeeks.com/ubuntu-20-04-vs-ubuntu-18-04-comparison-table/*

We are choosing a 64 bit x86 processor. For operating system, we are choosing Ubuntu, which is a Linux distribution. We will be using this operating system via a command line interface allowing us to type and execute Linux commands. Don't worry if you do not have prior experience with Linux. We will usually need very basic commands. I will provide a link below that can serve as a reference. You will find the system to be very intuitive and easy to understand.

Scrolling down this list of available images, you'll notice that outside the free tier you have options available for platforms suited to, for example, deep learning applications, with GPU support, etc. You can explore these options if your project requires something of that sort, and you'll need to look into the cost in that case. Your AWS educate credits will be useful here.

2. Choose Instance Type

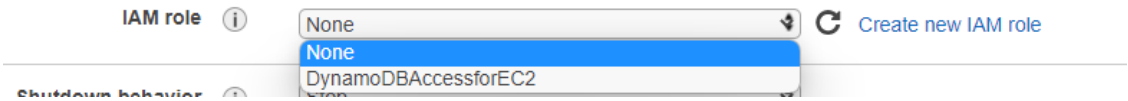
Here you specify the number of processors, amount of memory and networking capacity of your EC2 instance. We make a choice within the free tier, which is also the default choice.

☐	t2	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes

T2 instances are designed to be suitable for applications that require occasional (burst) spikes of computation or full core utilizations. For our purpose, t2.micro is more than sufficient. EBS (Elastic Block Storage) is a type of storage used for virtual data segments. In essence, this is a way to optimize performance by minimizing contention between your instance's I/O and other traffic.
This instance also provides IPv6 support, though in this lab I'll be using IPv4. A network capacity of low to moderate will suffice for our purposes.

3. Configure Instance Details
For the purpose of this lab, we do not need to make any changes to these configurations. However, in the next lab we will be using DynamoDB from our EC2 instance. So, let's select the option for that here.

Scroll down to the IAM role option and select DynamoDBAccessforEC2



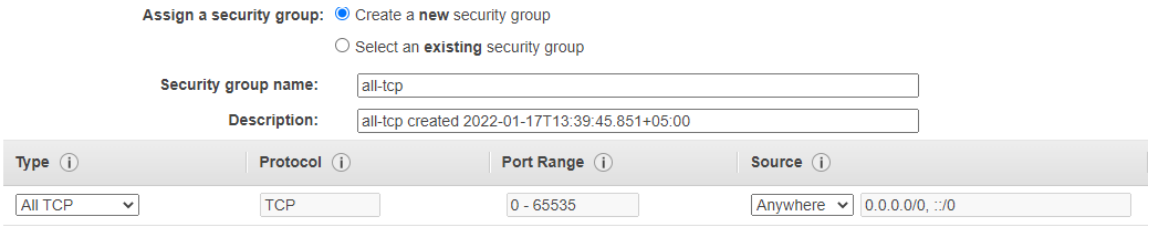
This means that our EC2 instance can access DynamoDB via this role.

4. Add Storage

In the Free Trier, we have 8GB of storage available. We'll leave that unchanged.

5. Add Tags
Tags are meta data that can be used to organize your resources. Various API calls can be used with tags to access resources in a systematic way. For now, we don't specify any tags.

6. Configure Security Group
This is an important piece of specification as it effects the ways in which we can access our instance (from our local machine, for example.)



As seen above, I have chosen to 'Create a new Security Group'. In security group name, I've written all-tcp. In the Type of security group below I have chosen All TCP. What this means in that any client operating over TCP will be able to establish a connection with our EC2 instance.

This is not the best choice in terms of security. Choosing SSH instead would require the client to use the secure SSH layer above TCP. However, selecting ALL TCP will allow our simple TCP client, without SSH, to communicate with our TCP server. 'ALL TCP' will also allow SSH clients like PuTTY to communicate with the server, as they too run over TCP. (But PuTTY would also work if we choose SSH instead of ALL TCP, because it uses SSH over TCP).

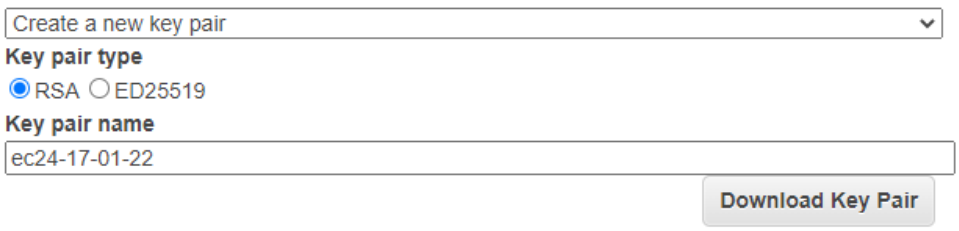
Later, if you wish to have an instance with SSH as the security group, you may add additional SSH specific code to your Python client.

In the source column, select Anywhere, so that any IP from anywhere can connect to the server. (Your local machine's IP will keep changing, so this is the right option for you.)

7. Review

After Pressing Review and Launch, you will be prompted to add a **key value pair**. This is a public/private key pair which is used by an encryption protocol, such as RSA. When connecting to our server through an SSH client, we will need to use this key-value pair, because SSH uses an encryption protocol when connecting to the server.

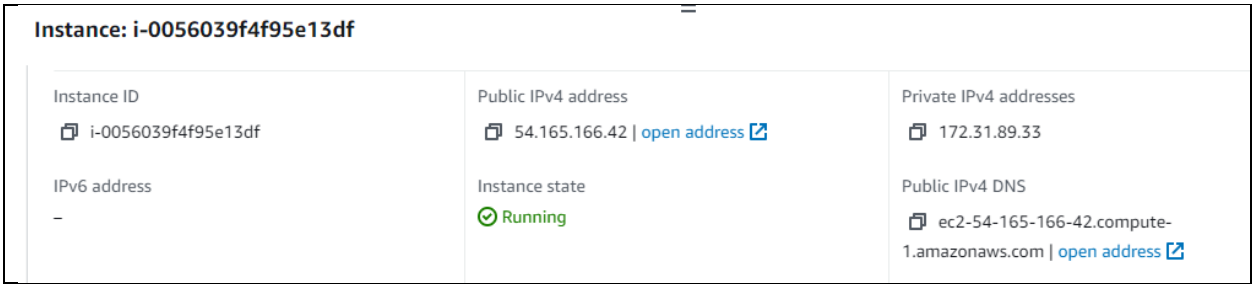
Therefore, we will need to provide the private key to our PuTTY client as well.



As shown above, we create a new key value pair, give it a useful name, and **Download Key Pair as a .pem file**.

After that, click Launch Instances. Your EC2 instance is launched and running.

From the EC2 dashboard you can access the details of this instance. Below are the details of the instance we just launched:

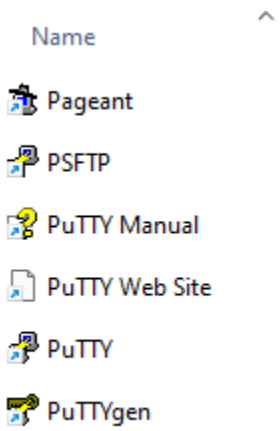


You will recall the difference between a public and private IP address from our discussion of NAT in Software Systems. For our local machine to connect to our EC2 instance, it can use either the public IPv4 address – in this case, 54.165.166.42; or, it can use the public IPv4 DNS, as we'll see below. If you stopped and relaunched this EC2 instance, it's public IP will change.

2.2 Communicating with the EC2 instance

Now we'll use **PuTTY**, an open-source terminal emulator operating over SSH, to communicate with our newly launched EC2 instance.

First, download and install PuTTY. Following components will be installed on your system:



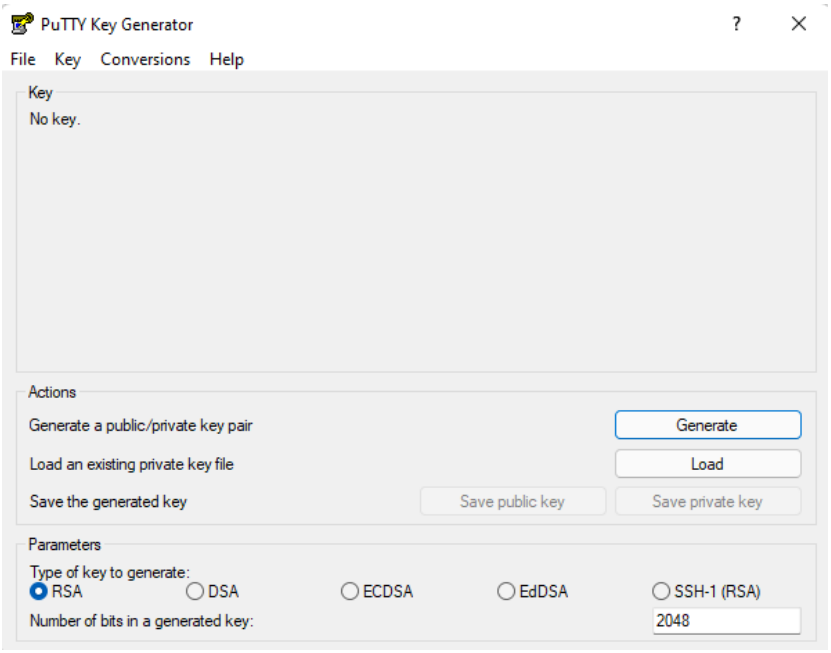
The **PuTTY** application (second from the bottom) is the terminal emulator that lets us run Linux commands on our Ubuntu based EC2 instance.

The **PuTTYgen** application lets us convert the .pem file (downloaded while instantiating ec2) into a .ppk file, PuTTY's format to store private keys.

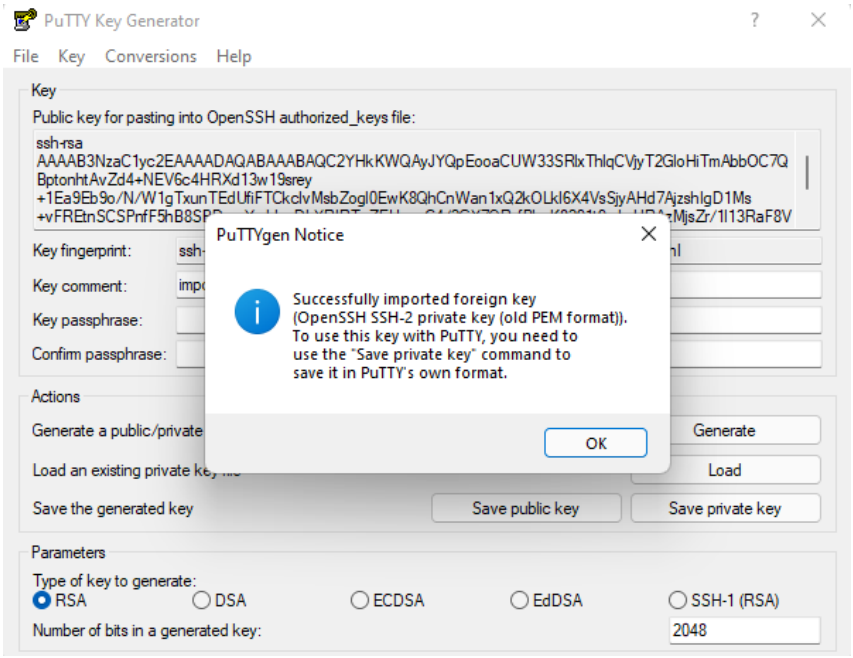
The **PSFTP** application is a secure SSH based file transfer tool. It is a command line tool which you can use to transfer files between your local computer and the server. However, we'll use another tool called **FileZilla** for this purpose. **FileZilla** provides a GUI for file transfer, making the process much simpler and speedier.

2.2.1 Use PuTTYgen to convert your .pem file to a .ppk file (PuTTY Private Key file)

- Run the PuTTYgen app.



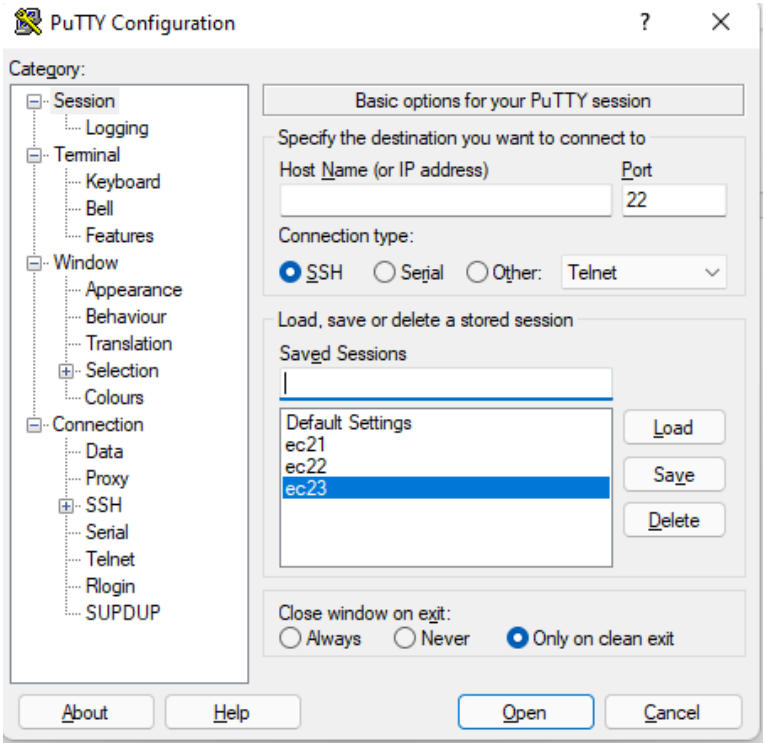
- Let the Type of key remain unchanged (i.e. RSA.)
- Click **Load** and select the downloaded .pem file from your system. (You'd need to change the file type option from *PuTTY Private Key Files (*.ppk)* to *All Files (*.*)* to see the .pem file.)
- Click Ok on the following dialog box



- Click 'Save private key' to save the key in the .ppk format.
- Close the PuTTYgen app.

2.2.2 Connect PuTTY to the EC2 instance

- Run the PuTTY app.

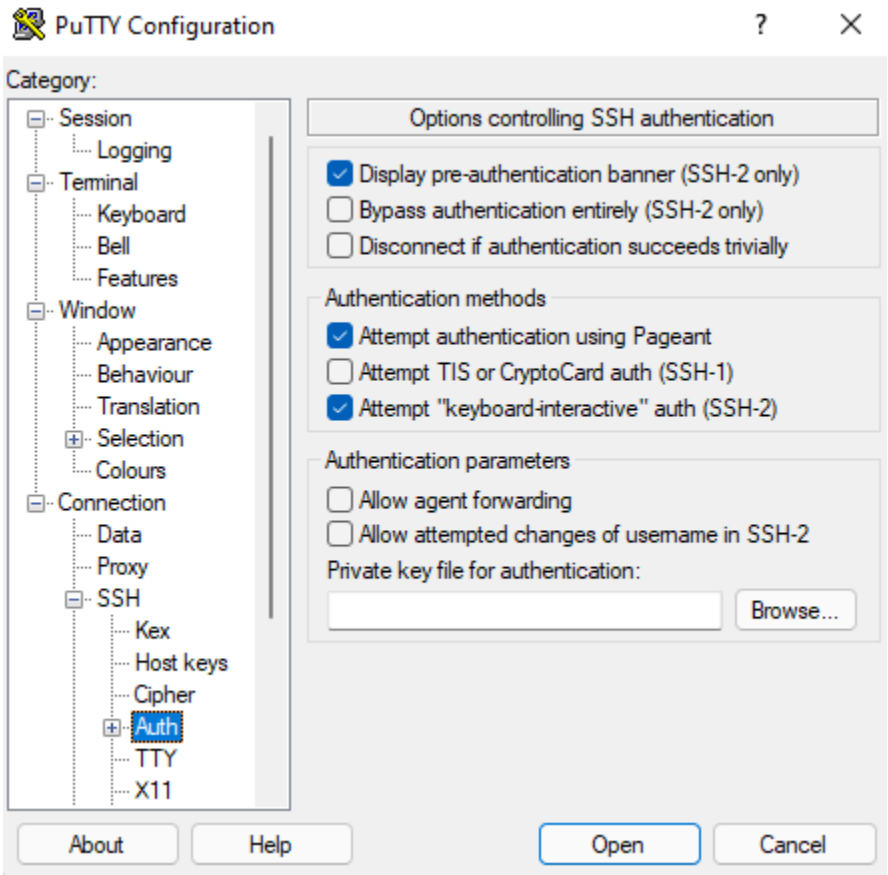


In the picture above, you see that I already have some sessions saved in PuTTY from previous use. You will save your first session shortly.

Notice the Port 22. This is port number used by the SSH protocol (the port number it inserts into the TCP socket).

- In the "Host Name (or IP address)" edit box you have the choice of entering either the public IP address of your EC2 instance – in our case, it was 54.165.166.42 -- or the Public IPv4 DNS name – in our case, it was ec2-54-165-166-42.compute-1.amazonaws.com.

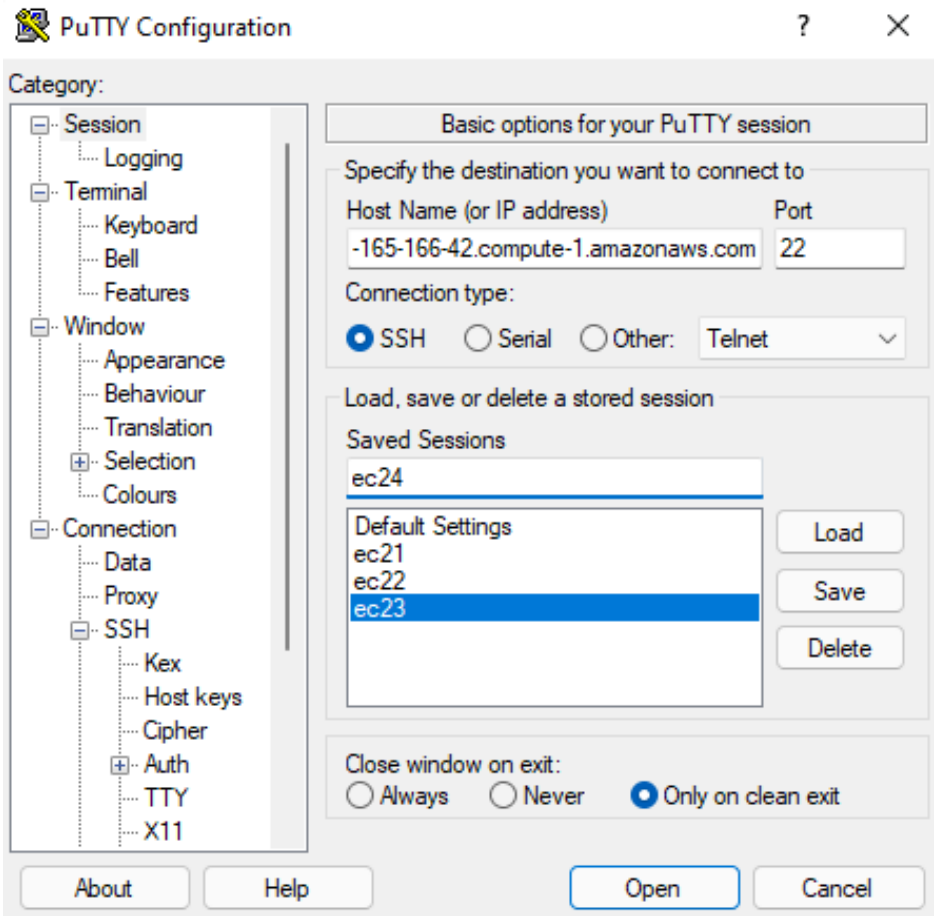
- From the Category pane on the left of the window, under Connection expand SSH and select Auth



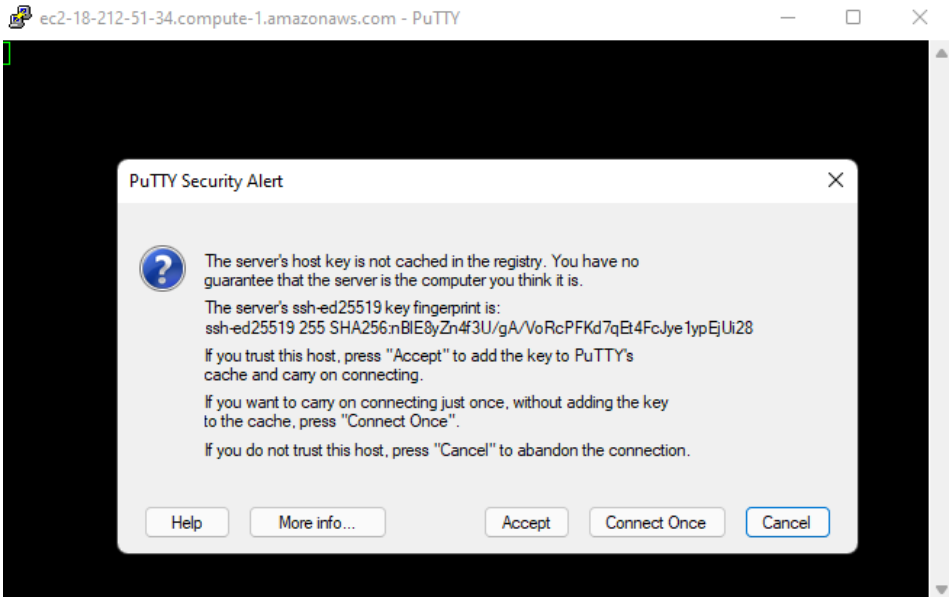
This is where you supply the private key .ppk file created in the last step.

- Click Browse and select the .ppk file.
- At this point, if we Click Open PuTTY will be connected to your EC2 instance. However, it is good idea to save our current connection settings (i.e. the current session) so that we can reconnect later using the same settings.

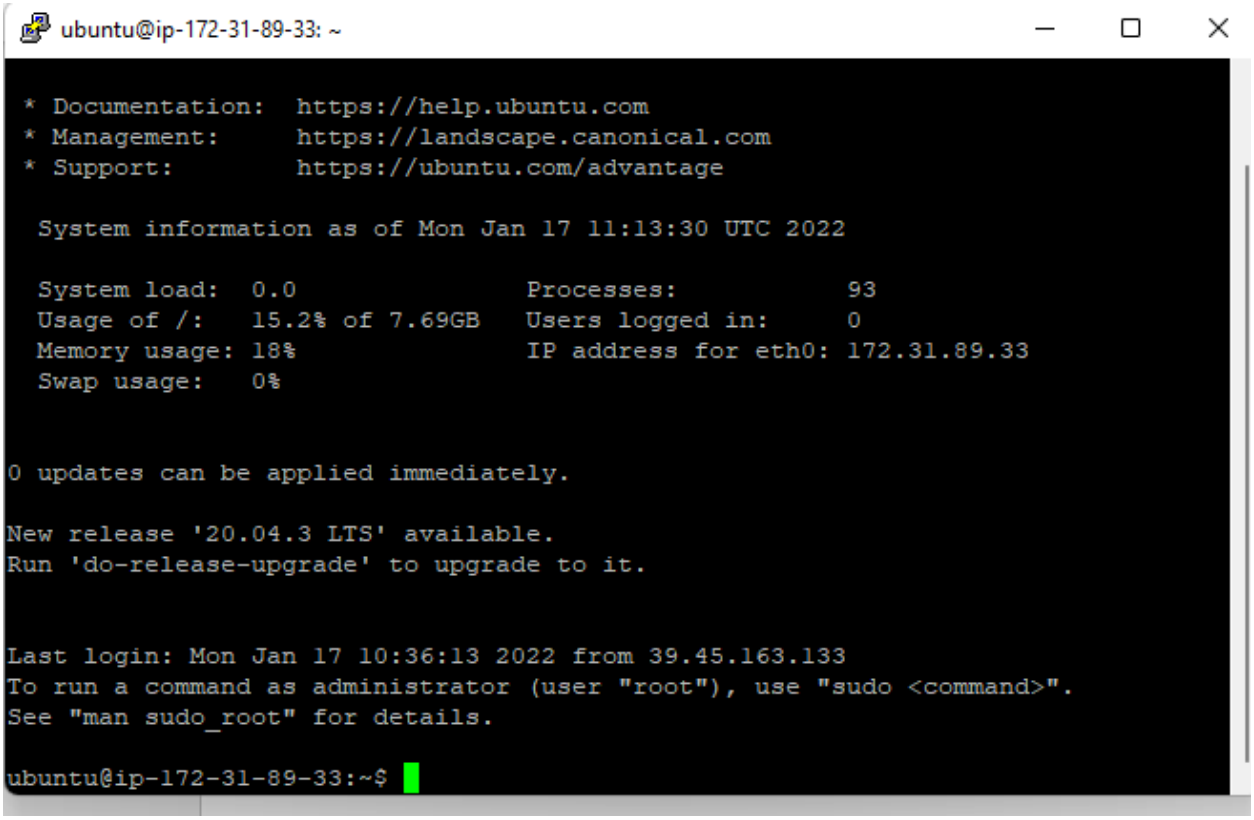
- Return to the Session window by selecting Session from the Category pane.
- Enter a session name (in the picture below I have chosen ec24 as the session name)
- Click save.
- Our current connection settings are now available as ec24 for later use.



- Click Open. You will see the following security alert. However, our server is trusted, so click Accept.



- In **login as:** enter ubuntu (this is your default username on the EC2 instance).
- Hit enter and you should be in your EC2 instance.



You can now execute Linux commands on the remote machine. Here is a link of some basic Linux commands. You'll find plenty of help on this on the Web.

- *For example, enter **ls -alF** to view all the files and directories in the current directory, including hidden files.*

```
drwxr-xr-x 5 ubuntu ubuntu 4096 Jan 17 10:36 ./
drwxr-xr-x 3 root   root   4096 Jan 17 09:04 ../
-rw-r--r-- 1 ubuntu ubuntu  220 Apr  4  2018 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu 3771 Apr  4  2018 .bashrc
drwx------ 2 ubuntu ubuntu 4096 Jan 17 10:36 .cache/
drwx------ 3 ubuntu ubuntu 4096 Jan 17 10:36 .gnupg/
-rw-r--r-- 1 ubuntu ubuntu  807 Apr  4  2018 .profile
drwx------ 2 ubuntu ubuntu 4096 Jan 17 09:04 .ssh/
```

*There is plenty of information here. If you further type **cd ..** you will go one level up into the home directory. Then type **ls -alF** again and you'll see the directory you were previously in, i.e., ubuntu. Doing a **cd ubuntu** will take you back inside the ubuntu directory. The columns of information in the graphic above show file permissions, number of links, owner name, owner group, file size in bytes, time of last modification, and file name.*

2.2.3 Install Python on the EC2 instance

This is a good time to install Python on our EC2 instance.

Simply entering the following on the command line should do the job:
$ sudo apt install python3.7
(when prompted, enter 'Y' and proceed)

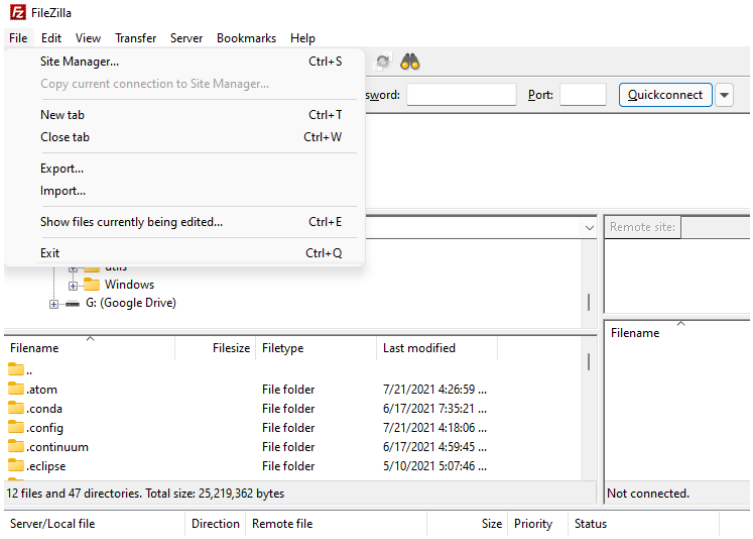
Once python is installed, you can check it by running:
$ python3
//this will take you inside Python terminal
>>> 3+5
///hitting enter will show you the result
>>> exit()
///hitting enter will exit from Python and take you back to Linux command line.

sudo is command that allows you to perform administrative tasks on a Linux system.
apt (Advanced Package Tool) is used to install various packages on the system. You can use it to install anything you need on the EC2 instance. Python here has been used as an example.

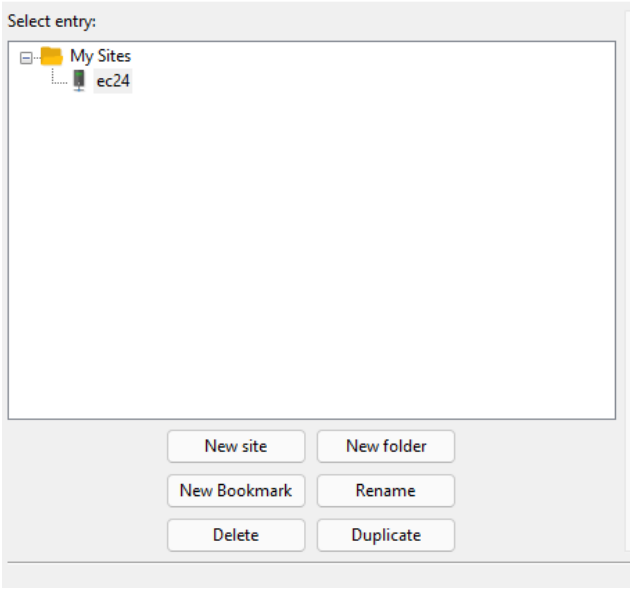
2.2.4 Move files to the server using FileZilla

With Python installed, we are now able to execute python code on our EC2 instance. We can write this code directly on the server using a Linux based editor like nano. However, it is much more convenient to first write the code on our local machine, then move it to EC2 and run it there. We need a way to transfer files from our local machine to EC2. We use WinSCP to accomplish this.

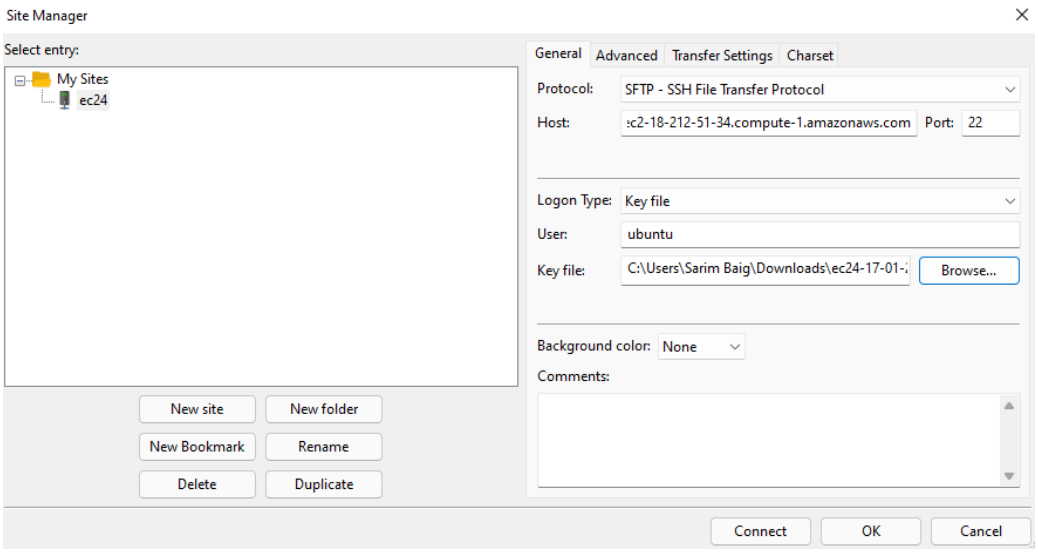
- First, download and install FileZilla.
- Run the FileZilla application. From the File menu select Site Manager.



- In Site Manager, click 'New site' and give a suitable name to the new entry. In the picture below I've given it the name ec24

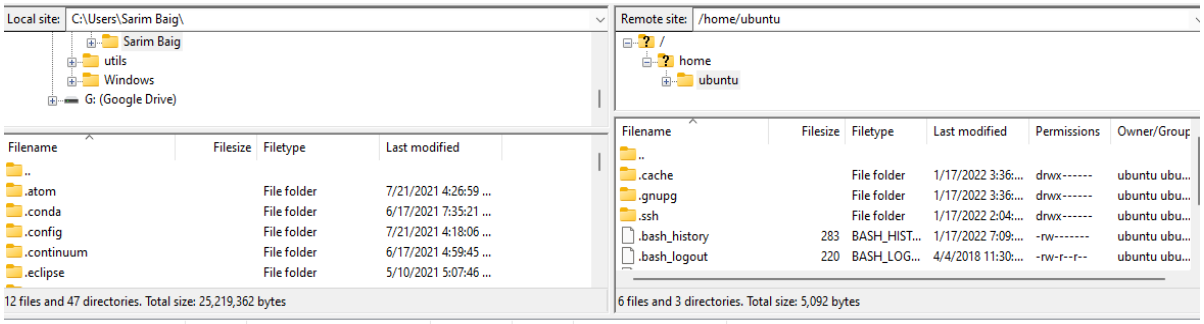


- Under the General tab, add information as shown below:



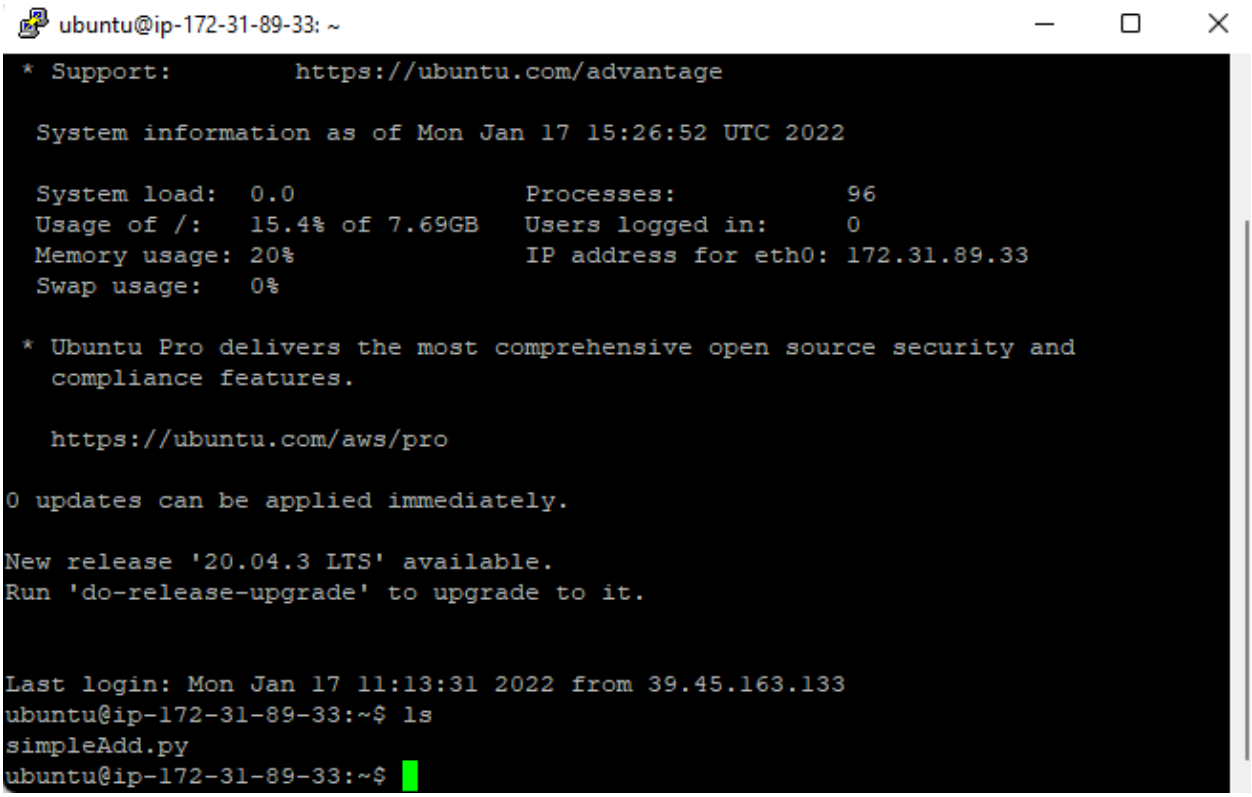
- In Protocol, choose SFTP – SSH File Transfer Protocol
- In Host, enter the Public IPv4 DNS of your EC2 instance.
- In Port write 22 (the SSH port)
- In the Logon Type, select Key file
- In User, enter ubuntu.
- For key file, Browse to the .ppk file created using PuTTYgen.
- Click Connect.

You will now see the local site (local computer) and remote site (EC2 instance) directory structure side by side. You can drag and drop files between them.



Move the file **simpleAdd.py** from your local machine to the EC2 instance. (This file has been provided with this guide.)

- Now let's return to PuTTY and list the contents in the current folder. You should see simpleAdd.py.



- To execute the program, enter:
  python3 simpleAdd.py

In the next section, we will move our familiar **tcpserver.py** file (from Software Systems) to our EC2 instance and run in there.

2.3 Running a TCP Server on the EC2 instance

We first set up the simple TCP server and client. Later, we run our server as a service on the EC2 instance so that even after our SSH session with the EC2 instance terminates the TCP server keeps running.

2.3.1 The Sever and the Client

You might recall the following code from your Software Systems class. This is a TCP server. You are familiar with the basic concepts of TCP sockets used here.

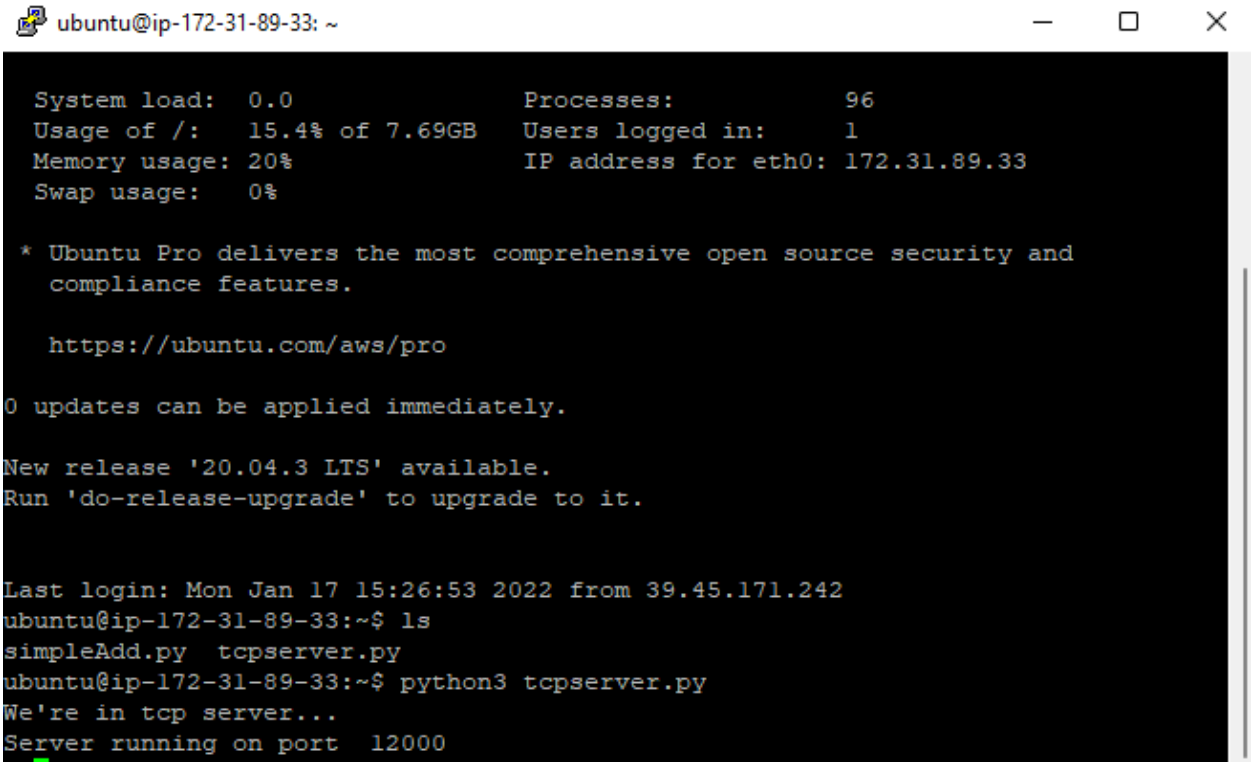
```
import socket
print("We're in tcp server...");

#select a server port
server_port = 12000
#create a welcoming socket
welcome_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
#bind the server to the localhost at port server_port
welcome_socket.bind(('0.0.0.0',server_port))

welcome_socket.listen(1)

#ready message
print('Server running on port ', server_port)

#Now the main server loop
while True:
   connection_socket, caddr = welcome_socket.accept()
   #notice recv and send instead of recvto and sendto
   cmsg = connection_socket.recv(1024)
   cmsg = cmsg.decode()
   if(cmsg.isalnum() == False):
      cmsg = "Not alphanumeric.";
   else:
      cmsg = "Alphanumeric";
   connection_socket.send(cmsg.encode())
```

Notice the IP address 0.0.0.0 used in the bind() function. When we wrote the same code on our local computer, we used the address 127.0.0.1 (i.e. 'localhost'). This is the IP address assigned to the "loopback" or local-only interface. This is a 'fake' network adapter that can only communicate within the same host. In comparison, when a server is told to listen on 0.0.0.0 that means 'listen on every available network interface', which is exactly what we want our remote server to do.

There is nothing special about the port number 12000. Any number outside the reserved space may be used.

- Copy this code to a python file, say *tcpserver.py*
- Move *tcpserver.py* to your EC2 instance using FileZilla
- Execute *tcpserver.py* on the EC2 instance.



The server is now running and ready to listen to a TCP client. We will run the client on our local machine. Following is the familiar python code for the TCP client.

```
import socket
print("We're in tcp client...");

#the server name and port client wishes to access
server_name = '18.212.51.34'

#'52.205.252.164'
server_port = 12000
#create a TCP client socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#Set up a TCP connection with the server
#connection_socket will be assigned to this client on the server side
client_socket.connect((server_name, server_port))

msg = input("Enter a string to test if it is alphanumeric: ");

#send the message  to the TCP server
client_socket.send(msg.encode())

#return values from the server
msg = client_socket.recv(1024)
```

```
print(msg.decode())
client_socket.close()
```

Once again, the thing to note is the IP address used in the connect function (stored in the server_name variable). This is the current Public IPv4 address of my EC2 instance. You should replace this address with the current Public IPv4 address of your instance.

Run this program on your computer. It should be able to communicate with the server.

2.3.1 Running the TCP Sever as a service

Currently, our TCP server will terminate as soon as our SSH session with the EC2 instance terminates. However, we will like to keep our server running in order to keep our application going. To keep our server running after the SSH session terminates (and to keep working on the terminal with the server running in the background) we need to launch our tcpserver.py code as a service (or a daemon, in Linux terminology).

This can be done in multiple ways. I describe just one, using systemd, a software component within Linux used to configure services.

Broadly, we will need to take the following steps:

Step 1: Configuring tcpserver.py as a service by creating a service configuration file

All services are located in /etc/systemd/system

Create a file, called tcpserver.service in this location. You can do this using the nano editor as follows:

~$ sudo nano /etc/systemd/system/tcpserver.service

The role of sudo is important here. We need super user rights to create the service file.

Add the following content to tcpserver.service

```
[Unit]
Description=TCP server service
After=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /home/ubuntu/tcpserver.py

[Install]
```

```
WantedBy=multi-user.target
```

This content is quite self-explanatory. The multi-user.target specifications indicate when after the system startup this service will be executed. ExecStart flag takes in the command that you want to run. The first argument is the python path and the second argument is the path to the script that needs to be executed.

Step 2: Launch the service

To launch the service, enter the following on the command line:

~$ sudo systemctl daemon-reload

~$ sudo systemctl enable tcpserver.service

~$ sudo systemctl start tcpserver.service

In these three lines, we have: reloaded the services information, enabled our service and started our service respectively. The TCP server is now running in the background as a service.

In order to confirm that it's there, enter the following on the command line:

~$ ps -ef

ps displays the currently running processes. You should see the tcpserver.py in the list.

Step 3: Communicate between client and server (now a service)

Simply run the tcpclient on your local machine. It should communicate with the server just as before.

Step 4 (How to) kill the server

At some stage you might want to kill the current server process before your launch an updated version. In order to do that, once again enter:

~$ ps -ef

Notice that the enter for tcpserver.py has a process id. This is the number in the second column in the tcpserver.py row. On my system at the moment this number is 25294. In order to kill the process, I will enter the following:

~$ sudo kill -9 25294

If you have multiply python processes running as services, you might want to save your time by doing

~$ sudo pkill python

This will terminate all python processes.

2.4 Exercises (beyond the lab)

(a) Compute the average RTT between your local computer (tcpclient.py) and the EC2 instance (tcpserver.py) while transmitting one integer of data. RTT is the time it takes for the integer to travel from the client to the server and back. Average this time over 500 communications and print the running average.

(b) tcpclient.py can accept data from our IoT devices and forward it to tcpserver.py, which can perform computations on it, and a so on. Add code to your tcpclient.py to accept accelerometer data from one of the FPGA boards. Send this data in string format to the tcpserver.py on EC2. The server should append each data unit with the prefix “Received:” and return it to the client. The client should print the output line by line.

***