TCSS 562: Software Engineering for Cloud Computing
Fall 2019
http://faculty.washington.edu/wlloyd/courses/tcss562

School of Engineering and Technology
University of Washington – Tacoma

Tutorial 10 – Cloud Service Performance Benchmarking

Disclaimer: Subject to updates as corrections are found
Version 0.10

The purpose of this tutorial is to leverage BASH script development to develop scripts to enable performance testing of IaaS cloud-hosted web services. The concepts in the tutorial build on concepts from tutorials #3 (Fall 2019) and #2 (Spring 2018). It is nessary to review and complete poritions of tutorial #2 from (Spring 2018) to successfully complete tutorial #10. Tutorial #2 (Spring 2018) is available at:
http://faculty.washington.edu/wlloyd/courses/tcss562_s2018/TCSS562_s2018_tutorial_2.pdf

The objective is to setup multiple client VMs to stress-test a Java webservice. The tutorial will make use of NTP, the network time protocol Linux service, to coordinate multiple client VMs, so they send requests at the same time. Additionally, this tutorial leverages pssh (parallel-SSH) to interact with multiple VMs in-parallel. We'll begin by launching two spot instances into a Virtual Private Cloud (VPCs), one to act as a client VM to author the test script, and one to act as the server.

To receive extra credit for this tutorial: From step 12, submit your CSV output file of your parallel webservice test to the professor via email. After using parallel-ssh to invoke the webservice test using multiple client VMs, concatenate all of the CSV output input a single CSV file and submit this file to Canvas.
cat outdir/* > all_output.csv

1. Adjusting your VPC Setup

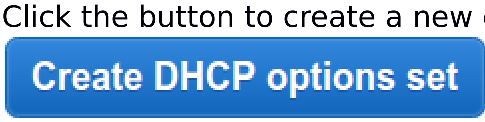
By default, instances launched into your VPC may display the following error message when "sudo" commands are run:

```
$ sudo w
sudo: unable to resolve host ip-10-0-0-195
```

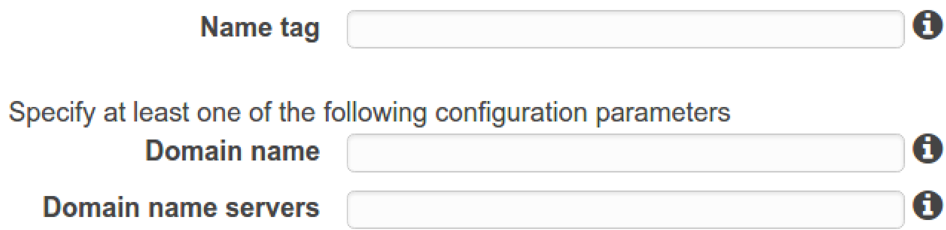
To solve this error, it is a best practice to ensure resolution of the fully qualified domain name (FQDN) for VMs. To ensure proper DNS resolution, configure your VPC's DHCP options set.

Resolve the error by setting the domain-name for the DHCP options set used by your VPC.

From the AWS management console, navigate to "VPC", and on the left-hand side select "DHCP Option Sets".
Click the button to create a new one:



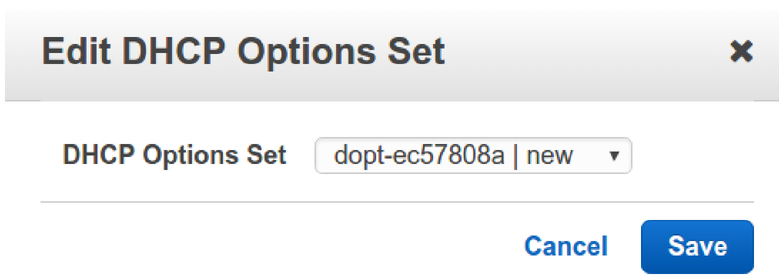
Populate the following:



Name tag: Give your new DHCP Options Set a name so you can easily identify it.
Domain name: "ec2.internal"
Domain name servers: "AmazonProvidedDNS"

For "Domain name", use ec2.internal if your region is US East (N. Virginia). For other regions, use region-name.compute.internal. For examples in us-west-2, use us-west-2.compute.internal. For the AWS GovCloud (US) region, use us-gov-west-1.compute.internal.

Once you've created the new option set, click on "Your VPCs" on the left, and select "Actions", "Edit DHCP Option Set". From the drop-down select your new DHCP Options Set:



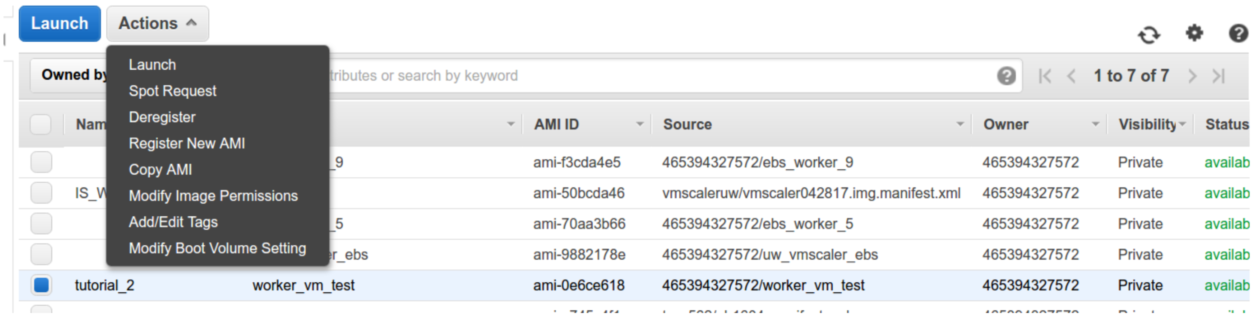
2. Create Spot Instances to start the Tutorial

After configuring your VPC's DNS behavior, next create two spot instances from your Apache Tomcat AMI created at the end of tutorial #2 (spring 2018).

Please note: It is suggested to start from a Ubuntu 16.04 base image to allow installation of tomcat7 using "sudo apt install tomcat7". Tomcat7 needs to be installed manually with Ubuntu 18.04+.

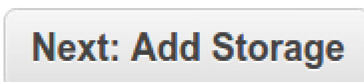
Follow steps in Spring 2018 tutorial #2 to create the Ubuntu 16.04 AMI.

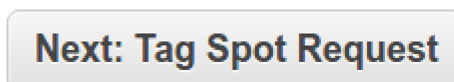
To save time (and charges) from setting up a NAT Gateway, let's assign these spot instances to use Public IPs.



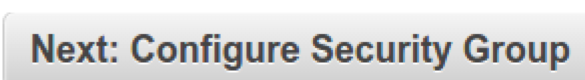
Use the following configuration settings:
For the instance type, choose any m4 or c4 series instance with a reasonable spot price in your availability zone such as m4/c4.large (2 vCPUs), m4/c4.xlarge (4 vCPUs), m4/c4.2xlarge (8 vCPUs), or m4/c4.4xlarge (16 vCPUs). **The recommended type is m4/c4.xlarge.** 5th generation VMs could alternatively be used, e.g. m5 or c5.xlarge.

Number of instances: 2
Maximum price: .50 (or alternate competitive price for your instance type)
Network: Choose your VPC created for tutorial #2 (spring 2018)
Subnet: Choose your public subnet created for tutorial #2 (spring 2018)
Auto-assign Public IP: Enable

Next, click the button: 

Accept the defaults for storage, click the button: 

Accept the defaults for tagging and click the button:



For Step 6, Configure Security Group, choose "select an existing security group", and then **<u>CHECK the box</u>** for: "default VPC security group" that was created for your VPC.

Next click the button: 

Review that your parameters are correct, and submit the spot request.

3. **Check settings on your new spot instances**

Once the VMs launch, ssh into one of the instances. Confirm that haproxy is not running.

```
$ sudo service haproxy status
```

In the output you should see:
Active: inactive (dead) since ……..

If you don't see this, stop haproxy:

```
$sudo service haproxy stop
```

Next, verify that Apache Tomcat7 is running

```
$ sudo service tomcat7 status
```

If tomcat7 is not running, start it:

```
$ sudo service tomcat7 start
```

Choose this VM to be the server, and the other VM to be the client.
Note the Public and Private IP address for each:

VM-Server IPs: ____
pub priv

VM-Client IPs: ____
pub priv

4. **Server VM setup**

On the VM-Server, let's install the Fibonacci web service application.

Navigate to "/var/lib/tomcat7/webapps":

```
cd /var/lib/tomcat7/webapps
```

Now grab the war file:

```
sudo wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/fibo.war
```

Watch the war file auto-deploy with the command:

```
watch -n 1 ls -alt
```

After a second or two, the web application will automatically be unpacked and a directory called "fibo" is automatically created.

Now, navigate to tomcat's logging directory:

```
cd /var/lib/tomcat7/logs
```

And tail the logfile:

```
tail -fn 30 catalina.out
```

5. **Client VM setup**

Now ssh into VM-Client.

Grab the test scripts (listed under "t4" from spring 2018):

```
wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibonacci.sh
```

Now, using vi or pico, edit the test script.
In the script, update host=xx.xx.xxx.xxx to be the "Private IP" address of your VM-Server:
host=xx.xx.xxx.xxx

Do not use the IPv4 Public IP.

Set the script so your user account has execute permission

```
chmod u+x testFibonacci.sh
```

Now, try running the script as follows:

```
time ./testFibonacci.sh
```

By default, the tomcat7 configuration limits heap memory to ~128 MB. This limits the size of the fibonacci number we can process.

6. **Increase Tomcat server memory**

Next, update the tomcat7 configuration to boost the heap size.

Go to the tomcat bin directory:

```
cd /usr/share/tomcat7/bin
```

Grab the setenv.sh file:

```
sudo wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/setenv.sh
```

Set the file to have execute permission:

```
sudo chmod a+x setenv.sh
```

The setenv.sh sets environment variables for tomcat7. It is invoked by catalina.sh.

And now restart tomcat7 to load the configuration change.

```
sudo service tomcat7 restart
```

On the server, once again trace the tomcat7 logfile output:

```
tail -fn 30 /var/lib/tomcat7/logs/catalina.out
```

7. **Test a large fibonacci request**

Now edit testFibonacci.sh to change the size of the fibonacci to “300000”.

By increasing tomcat7’s Java heap size, we can request a larger fibonacci number to make the service more severely memory and compute bound.

When piped to a JSON file, Fibonacci 300000 generates a JSON file that is 62,755 bytes! That’s a BIG NUMBER! The number has 62,698 digits!

Generating this fibonacci result takes several seconds on a c4.xlarge:

```
real    0m3.043s
user    0m0.008s
sys     0m0.000s
```

Running “top -d .2” on VM-Server, you see how this service request stresses the VM:

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12267 tomcat7   20   0 11.420g 6.649g  16104 S 180.0 91.1   0:49.81 java
```

91% of available memory and **180% of CPU capacity** is utilized at one point !
It will not be possible to run parallel requests without crashing tomcat7 because the VM will run out of memory.

8. **Install and configure the NTP service on the client VM**

Let’s now install the Network Time Protocol (NTP) service and related programs:

```
sudo apt install ntp ntpdate ntpstat
```

Select “Y” to install.
NTP starts up automatically. First we need to manually shutdown NTP, set the time against a NTP server, then restart ntp.

```
sudo service ntp stop
```

The “ntpdate” utility allows us to directly set the time against a NTP server. This immediately sets the time to match the server as opposed to ntpd which only slowly synchronizes clocks which do not match.

```
sudo ntpdate 0.amazon.pool.ntp.org
```

21 May 07:48:03 ntpdate[2654]: adjust time server 216.6.2.70 offset -0.001937 sec

Next, edit the /etc/ntp.conf file to specify the NTP servers to use for clock synchronization:

```
pico /etc/ntp.conf
```

While the default ubuntu ntp servers configured may suffice, let’s switch to use the AWS servers.

Under the “# Specify on or more NTP servers” comment, replace the NTP servers with:

```
server 0.amazon.pool.ntp.org iburst
server 1.amazon.pool.ntp.org iburst
server 2.amazon.pool.ntp.org iburst
server 3.amazon.pool.ntp.org iburst
```

Now, start the ntp service:

```
sudo service ntp start
```

Use the ntpstat command to test ntp:

```
ntpstat
```

Next, ensure that ntp will start automatically when the system boots up by issuing setting the services to have the default service startup settings with sysv-rc-conf:

```
sudo apt install sysv-rc-conf
sudo sysv-rc-conf ntp on
sudo sysv-rc-conf --list ntp
```

sysv-rc-conf will show what “runlevels” the service starts on:

ntp 1:off 2:on 3:on 4:on 5:on

To learn more about runlevels in Ubuntu Linux, see this blog:
http://www.pathbreak.com/blog/ubuntu-startup-init-scripts-runlevels-upstart-jobs-explained

The “chkconfig” utility appears no longer available in default repositories for Ubuntu 16.04, so sysv-rc-conf is used instead.

9. **Create a BASH testing script to stress test the service**

Next, we will make changes to our basic bash script to allow it to be more robust for load testing.

1. Install tools for the script

We will want to install the “bc” utility onto the client:

```
sudo apt install bc
```

“bc”, is a “calculator for bash that allows us to do basic addition, subtraction, division, etc. We use the “echo” command to print numbers and then pipe them to bc, and bc grabs them to perform a calculation.

We will also install “GNU parallel”, a Linux utility which supports running multiple tasks in parallel. While tasks can be “backgrounded” with the &, GNU parallel has some nice features to coordinate running tasks in parallel.

```
sudo apt install parallel
```

2. Add a for loop

A for loop will allow a specified number of repeated tests. We will use the first command line argument to provide the duration of the for loop.

Change the script as follows:
This script can be downloaded as:

```
wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibLoop.sh
```

```
host=10.0.0.124
port=8080
runs=$1

for (( i=1 ; i <= $runs; i++ ))
do
  json={"\"number\"":300000}
  echo "test $i: $json"
      curl -X POST -H "Content-Type: application/json" http://$host:$port/fibo/fibonacci -d $json
  echo ""
  sleep .2
done
```

3. Replace with an timed sleep call

Next use the refactored script. This version adds a sleep calculation to send approximately 1 request per second. The service turnaround time is captured and CSV output is generated:

This script can be downloaded as:

```
wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibProporSleep.sh
```

Script appears on next page:

```
#!/bin/bash
host=10.0.0.124
port=8080
runs=$1
onesecond=1000

echo "run_id,json,elapsed_time,sleep_time_ms"
for (( i=1 ; i <= $runs; i++ ))
do
  json={"\"number\"":50000}
  time1=( $(($(date +%s%N)/1000000)) )
     curl  -X  POST  -H  "Content-Type:  application/json"  http://$host:
$port/fibo/fibonacci -d $json >/dev/null 2>/dev/null
  time2=( $(($(date +%s%N)/1000000)) )
  elapsedtime=`expr $time2 - $time1`
  sleeptime=`echo $onesecond - $elapsedtime | bc -l`
  sleeptimems=`echo $sleeptime/$onesecond | bc -l`
  echo "Run-$i,$json,$elapsedtime,$sleeptimems"
  if (( $sleeptime > 0 ))
  then
    sleep $sleeptimems
  fi
done
```

The service turnaround time for each request is reported in ms.
This script makes a synchronous service request. The bash "thread" stays active so the turnaround time can be captured using the system timer. As long as the turnaround time for a single service request is less than one second, this client will sustain a 1 request per second load.

4. Use a function and GNU parallel to generate load requests in parallel

Next, we'll refactor the script to run multiple threads in parallel which generate 1 request per second. We'll move the previous code into the "callservice()" function and use GNU parallel to invoke separate instances of callservice() in parallel with different threads.

The "export" command, exports the function so it can be called from another bash shell...

This script can be downloaded as:

wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibPar.sh

```
#!/bin/bash
totalruns=$1
threads=$2

callservice() {
  totalruns=$1
  threadid=$2
  host=10.0.0.124
```

```
  port=8080
  onesecond=1000
  if [ $threadid -eq 1 ]
  then
    echo "run_id,thread_id,json,elapsed_time,sleep_time_ms"
  fi
  for (( i=1 ; i <= $totalruns; i++ ))
  do
    json={"\"number\"":50000}
    time1=( $(($(date +%s%N)/1000000)) )
    curl -X POST -H "Content-Type: application/json" http://$host:$port/fibo/fibonacci -d $json >/dev/null 2>/dev/null
    time2=( $(($(date +%s%N)/1000000)) )
    elapsedtime=`expr $time2 - $time1`
    sleeptime=`echo $onesecond - $elapsedtime | bc -l`
    sleeptimems=`echo $sleeptime/$onesecond | bc -l`
    echo "$i,$threadid,$json,$elapsedtime,$sleeptimems"
    if (( $sleeptime > 0 ))
    then
      sleep $sleeptimems
    fi
  done
}
export -f callservice

runsperthread=`echo $totalruns/$threads | bc -l`
runsperthread=${runsperthread%.*}
echo "Setting up test: runsperthread=$runsperthread threads=$threads totalruns=$totalruns"
for (( i=1 ; i <= $threads ; i ++))
do
  arpt+=($runsperthread)
done
parallel --no-notice -j $threads -k callservice {1} {#} ::: "${arpt[@]}"
```

Now try out this script.

For 1 thread, 1 run in parallel:

```
./testFibonacci.sh 1 1
```

Setting up test: runsperthread=1 threads=1 totalruns=1
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},71,.92900000000000000000

For 2 threads, 2 runs in parallel:

```
./testFibonacci.sh 2 2
```

Setting up test: runsperthread=1 threads=2 totalruns=2
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},145,.85500000000000000000
1,2,{"number":50000},144,.85600000000000000000

For 3 threads, 3 runs in parallel:

```
./testFibonacci.sh 3 3
```

Setting up test: runsperthread=1 threads=3 totalruns=3
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},177,.82300000000000000000
1,2,{"number":50000},140,.86000000000000000000
1,3,{"number":50000},177,.82300000000000000000

For 4 threads, 4 runs in parallel:

```
./testFibonacci.sh 4 4
```

Setting up test: runsperthread=1 threads=4 totalruns=4
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},317,.68300000000000000000
1,2,{"number":50000},319,.68100000000000000000
1,3,{"number":50000},328,.67200000000000000000
1,4,{"number":50000},333,.66700000000000000000

For 5 threads, 5 runs in parallel:

```
./testFibonacci.sh 5 5
```

Setting up test: runsperthread=1 threads=5 totalruns=5
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},193,.80700000000000000000
1,2,{"number":50000},340,.66000000000000000000
1,3,{"number":50000},347,.65300000000000000000
1,4,{"number":50000},365,.63500000000000000000
1,5,{"number":50000},374,.62600000000000000000

For 6 threads, 6 runs in parallel:

```
./testFibonacci.sh 6 6
```

Setting up test: runsperthread=1 threads=6 totalruns=6
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},457,.54300000000000000000
1,2,{"number":50000},472,.52800000000000000000
1,3,{"number":50000},455,.54500000000000000000
1,4,{"number":50000},641,.35900000000000000000
1,5,{"number":50000},455,.54500000000000000000
1,6,{"number":50000},465,.53500000000000000000

For 7 threads, 7 runs in parallel:

```
./testFibonacci.sh 7 7
```

Setting up test: runsperthread=1 threads=7 totalruns=7
run_id,thread_id,json,elapsed_time,sleep_time_ms
1,1,{"number":50000},426,.57400000000000000000
1,2,{"number":50000},454,.54600000000000000000
1,3,{"number":50000},435,.56500000000000000000
1,4,{"number":50000},477,.52300000000000000000
1,5,{"number":50000},436,.56400000000000000000
1,6,{"number":50000},474,.52600000000000000000

1,7,{"number":50000},483,.51700000000000000000

Note how increasing the number of threads in parallel is causing the elapsed time to increase. A single run only takes 71ms. But doing 7 in parallel each takes about 426ms.

5. Add parameter to specify fibonacci number

Now we'll add a 3rd parameter to the script to enable the fibonacci number to be passed in. If the parameter is omitted, or if a 0 is provided we'll use a random number from "1 to $RANDOM * 10". In BASH, $RANDOM provides a random number from 0 to 32767.

This script can be downloaded as:

wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibParAdj.sh

Changes to the script are in yellow below:

```
#!/bin/bash
totalruns=$1
threads=$2
fibo=$3

callservice() {
  totalruns=$1
  threadid=$3
  fibonum=$2
  host=10.0.0.124
  port=8080
  onesecond=1000
  #echo "args 1=$1 2=$2 3=$3"
  if [ $threadid -eq 1 ]
  then
    echo "run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum"
  fi
  if [ $fibonum == "\"\"" ] || [ $fibonum == "\"0\"" ]
  then
    fibonum=`echo "${RANDOM}0"`
    #echo "Random fibo $fibonum"
  fi
  for (( i=1 ; i <= $totalruns; i++ ))
  do
    json={"\"number\"":$fibonum}
    time1=( $(($(date +%s%N)/1000000)) )
          curl  -X  POST  -H  "Content-Type:  application/json"  http://$host:
$port/fibo/fibonacci -d $json >/dev/null 2>/dev/null
    time2=( $(($(date +%s%N)/1000000)) )
    elapsedtime=`expr $time2 - $time1`
    sleeptime=`echo $onesecond - $elapsedtime | bc -l`
    sleeptimems=`echo $sleeptime/$onesecond | bc -l`
    echo "$i,$threadid,$json,$elapsedtime,$sleeptimems,$fibonum"
    if (( $sleeptime > 0 ))
    then
      sleep $sleeptimems
```

```
    fi
  done
}
export -f callservice

runsperthread=`echo $totalruns/$threads | bc -l`
runsperthread=${runsperthread%.*}
echo "Setting up test: runsperthread=$runsperthread threads=$threads totalruns=$totalruns fibonum=$fibo"
for (( i=1 ; i <= $threads ; i ++))
do
  arpt+=($runsperthread)
done
afibo="\"$fibo\""
parallel --no-notice -j $threads -k callservice {1} {2} {"#"} ::: "${arpt[@]}" ::: "${afibo[@]}"
```

6. Synchronize start of the script using time comparison

Since the client VMs have NTP, their clocks are synchronized.
We can add a loop to the start of the script to check if we've reached a specified start time. Using this approach, a request can be made to start scripts across multiple client VMs at a synchronized start time by checking the system time.

At the top of the script, add a 4th command line argument:

```
starttime=$4
```

Now, add the following block of code after "export -f callservice", but before setting up the parallel service calls:

```
# If a starttime is provide, loop until we reach the start time before calling service
if [ ! -z "$starttime" ]
then
  t1=`date --date="$starttime" +%s`
  echo "Start script at $t1"
  while : ; do
    dt2=`date +%Y-%m-%d\ %H:%M:%S`
    # Compute the seconds since epoch for date 2
    current_time=`date --date="$dt2" +%s`
    sleep .1
    #echo "compare $current_time >= $t1"
    [ "$current_time" -lt "$t1" ] || break
  done
echo "Starting script now... $current_time"
fi
```

This script can be downloaded as:

wget http://faculty.washington.edu/wlloyd/courses/tcss562/tutorials/t4/testFibParTime.sh

If the 4th argument, the starttime, is not provided, the "synchronized" sleep loop is skipped altogether. To simplify time comparisons, we convert times to seconds after the epoch (January 1, 1970). This way, the date/time comparison just compares two integers as opposed to a complicated date string.

Please specify the date/time in "yyyy-mm-dd hh:mm:ss" format. The date/time **MUST be in double quotes.**

Check the current date

```
date
```

Mon May 22 06:53:30 UTC 2017

Then choose a date/time in the future. The testFibonacci script will "spin" and wait to start making web service requests until the specified time.

```
./testFibonacci.sh 3 3 3000 "2017-05-22 06:54:10"
```

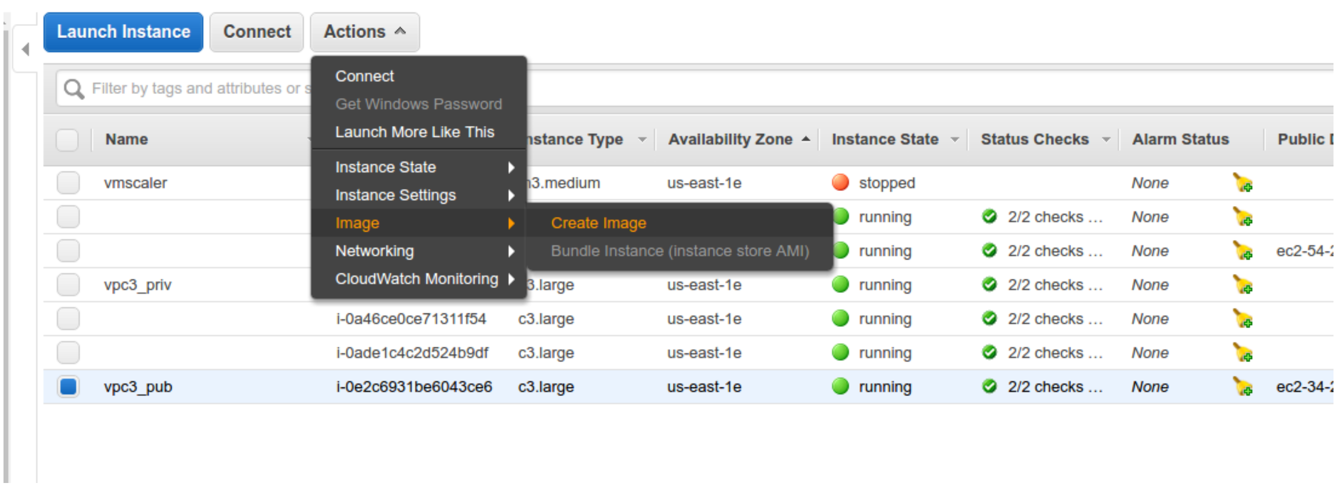
Start script at 1495436050 2017-05-22 06:54:10
Starting script now... 1495436050
Setting up test: runsperthread=1 threads=3 totalruns=3 fibonum=3000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"3000"},11,.98900000000000000000,"3000"
1,2,{"number":"3000"},8,.99200000000000000000,"3000"
1,3,{"number":"3000"},8,.99200000000000000000,"3000"

This "timed" start is useful to help synchronize a large number of clients to simultaneously run commands across multiple VMs **at the same time**.

10. **Make images of the client and server VMs**

Next, create images of the client VM, so it can be replicated…
As in tutorial 2 (spring 2018), we'll image the EBS root backed instances (these are "hvm")

Select your client instance and click Actions, "Create Image"



In the “Create Image” dialog, provide:

Image name: client_vm

Leave all other options as-is.

Click “Create Image”…

An image of your instance with your testing script and newly installed packages is created after a few minutes.

Optionally, you may now also create an image of your server VM at this time for backup purposes.

Image name: server_vm

Once images are created, you can shutdown your spot instances to resume working on the tutorial later on to save costs.

11. **Launch additional client VMs**

To generate requests from multiple client VMs, let’s create more spot instances using your newly created client image. Launch at least 3 or more additional client VMs of the same m4 instance type as your existing client.

12. **Configure pssh on one client VM**

Install the parallel-ssh client on your original client VM.

```
sudo apt install pssh
```

Once the utility is installed create a hosts file which contains the list of client VM IP addresses. Name this file “hosts” for example:

<u>example hosts file</u>

```
cat hosts
```

10.0.0.126:22
10.0.0.225:22
10.0.0.69:22
10.0.0.181:22

You can *include* your client VM running pssh as one of the hosts.

With the hosts file defined, create an outdir and errdir for output and error files.

```
mkdir outdir
mkdir errdir
```

Next, parameterize the "parallel-ssh" command as follows:

```
parallel-ssh -h hosts -l ubuntu -x "-oStrictHostKeyChecking=no -i uw_wlloyd_1.pem" -o outdir -e errdir
'./testFibonacci.sh 6 3 30000 "2017-05-22 06:48:00"'
```

-h: specifies the host file name
-l: spceifies the username
-x: passes quoted elements to ssh, allows the keyfile to be specified
-o: specifies an output directory for output files
-e: specifies an error directory for error files

To specify a start date/time, you must first have the testFibonacci script in single quotes, and the date/time in double quotes.

This setup will allow you to launch service requests in parallel across your worker client VMs.

To display output use the following command:

```
cat outdir/*
```

13. Comparing service performance with increasing workloads

Now, here is the output of a test for 1-VM with 3 threads, running 3 tests of 30,000:

```
$ ./testFibonacci.sh 6 3 30000 "2017-05-22 07:08:00"Start script at 1495436880 2017-05-22
07:08:00
```

Starting script now... 1495436880
Setting up test: runsperthread=2 threads=3 totalruns=6 fibonum=30000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"30000"},32,.96800000000000000000,"30000"
2,1,{"number":"30000"},34,.96600000000000000000,"30000"
1,2,{"number":"30000"},54,.94600000000000000000,"30000"
2,2,{"number":"30000"},42,.95800000000000000000,"30000"
1,3,{"number":"30000"},43,.95700000000000000000,"30000"
2,3,{"number":"30000"},42,.95800000000000000000,"30000"

Here you see the requests took 32, 34, 54, 42, 43, 42 ms respectively.
The average is 41.166 ms.

Now if we run 4 VMs in parallel against the same server synchronized to start at the same time, note how the performance degrades.

The average turnaround time for the service increases to 61.58 ms per service request.

```
$ parallel-ssh -h hosts -l ubuntu -x "-oStrictHostKeyChecking=no -i uw_wlloyd_1.pem" -o outdir -e errdir
'./testFibonacci.sh 6 3 30000 "2017-05-22 07:09:00"'
```

[1] 07:09:02 [SUCCESS] 10.0.0.181:22
[2] 07:09:02 [SUCCESS] 10.0.0.126:22
[3] 07:09:03 [SUCCESS] 10.0.0.225:22
[4] 07:09:03 [SUCCESS] 10.0.0.69:22

```
ubuntu@ip-10-0-0-181:~$ cat outdir/*
```

```
Start script at 1495436940
Starting script now... 1495436940
Setting up test: runsperthread=2 threads=3 totalruns=6 fibonum=30000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"30000"},53,.947000000000000000,"30000"
2,1,{"number":"30000"},77,.923000000000000000,"30000"
1,2,{"number":"30000"},44,.956000000000000000,"30000"
2,2,{"number":"30000"},71,.929000000000000000,"30000"
1,3,{"number":"30000"},53,.947000000000000000,"30000"
2,3,{"number":"30000"},75,.925000000000000000,"30000"
Start script at 1495436940 2017-05-22 07:09:00
Starting script now... 1495436940
Setting up test: runsperthread=2 threads=3 totalruns=6 fibonum=30000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"30000"},42,.958000000000000000,"30000"
2,1,{"number":"30000"},69,.931000000000000000,"30000"
1,2,{"number":"30000"},31,.969000000000000000,"30000"
2,2,{"number":"30000"},88,.912000000000000000,"30000"
1,3,{"number":"30000"},41,.959000000000000000,"30000"
2,3,{"number":"30000"},86,.914000000000000000,"30000"
Start script at 1495436940
Starting script now... 1495436940
Setting up test: runsperthread=2 threads=3 totalruns=6 fibonum=30000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"30000"},42,.958000000000000000,"30000"
2,1,{"number":"30000"},32,.968000000000000000,"30000"
1,2,{"number":"30000"},31,.969000000000000000,"30000"
2,2,{"number":"30000"},44,.956000000000000000,"30000"
1,3,{"number":"30000"},41,.959000000000000000,"30000"
2,3,{"number":"30000"},44,.956000000000000000,"30000"
Start script at 1495436940
Starting script now... 1495436940
Setting up test: runsperthread=2 threads=3 totalruns=6 fibonum=30000
run_id,thread_id,json,elapsed_time,sleep_time_ms,fibonum
1,1,{"number":"30000"},95,.905000000000000000,"30000"
2,1,{"number":"30000"},89,.911000000000000000,"30000"
1,2,{"number":"30000"},90,.910000000000000000,"30000"
2,2,{"number":"30000"},80,.920000000000000000,"30000"
1,3,{"number":"30000"},69,.931000000000000000,"30000"
2,3,{"number":"30000"},91,.909000000000000000,"30000"
```

One interesting result I observed was if I sychronized a batch of repeating identical fibonacci service requests to have a synchronized start, with synchronized sleeps,

performance was actually better, than with a random start. I suspect that when service requests occur *close-in-time* system caching is helping them run faster in this manner.

14. **Reusing the scripts**

The testFibonacci.sh script can be refactored to test **ANY** cloud service, including AWS Lambda functions.
Simply edit the callservice() function to perform desired work.
This could be a service request using curl, or any activity using various Linux cloud command-line interfaces. For example CLIs for redis, mysql, postgresql, can be invoked.

The test script provides the basis to:

1. Perform a synchronized service requests across multiple client VMs in parallel
2. Using GNU parallel, invoke the callservice() function multiple times in parallel on a single VM
3. Generate CSV output capturing turnaround time of service requests

15. **Updating script files across client VMs**

Parallel-scp is similar to parallel-ssh and allows parallel file transfers to the VMs defined in your hosts file.

Here's the syntax to transfer a test.sh file.

Parallel-scp requires the client destinations to have a fully-qualified path:

```
parallel-scp -h hosts -l ubuntu -x "-oStrictHostKeyChecking=no -i uw_wlloyd_1.pem" test.sh /home/ubuntu/test.sh
```

HOW TO SUBMIT THIS TUTORIAL

To receive credit for the tutorial, concatenate all of the output into a single CSV file, and submit this CSV on Canvas:

```
cat outdir/* > all_output.csv
```

16. **Cleanup**

At the end of the tutorial, reimage your client VM with pssh installed if this would be helpful for testing with pssh for the term project.

Next, **TERMINATE** all EC2 instances, and **DELETE** unneeded EBS volumes. Also, be sure to **DELETE your NAT Gateway**, and **DELETE any Elastic IP addresses** that were created.

Failing to delete all cloud artifacts will result in loss of AWS credits leading to potential charges to your credit card.

As a reminder (again), when concluding, please:

1. **DELETE your NAT gateway**
2. **DELETE any Elastic IP addresses**

Failing to do so will result in charges.

******************** **WARNING ABOUT NAT GATEWAYS** ********************

NAT Gateway's are special virtual appliances that provide gateway/router functionality between your subnetworks of the VPC and the internet.

IT IS ESSENTIAL TO DELETE NAT GATEWAYS AFTER COMPLETING THE TUTORIAL.

NAT Gateway costs for US-EAST-1:

4.5 cents / hour
$1.08/day
$7.56/week
$32.40/month
$388.80/year

INSTRUCTIONS TO DELETE NAT GATEWAY:

Under the VPC Dashboard:

On the left-hand sidebar, select "NAT Gateways".
Select the ROW in the table for your NAT Gateway.
Click the "Actions" button, then select "Delete NAT Gateway"

FAILURE TO DELETE YOUR NAT GATEWAY
WILL RAPIDLY DEPLETE YOUR AWS CREDITS !!!!

****