Tutorial 9 – Asynchronous Function Profiling with SAAF

Disclaimer: Subject to updates as corrections are found
Version 0.11
Scoring: 20 pts maximum

The purpose of this tutorial is to introduce asynchronous function profiling with the Serverless Application Analytics Framework (SAAF) in AWS Lambda and Java.

This tutorial also provides a source code example of calling an AWS Lambda function from a AWS Lambda function. This tutorial also provides a source code example of publishing SAAF data to an S3 bucket. This tutorial also provides **optional activities**: step-by-step instructions on: (1) how to set-up a NAT gateway for a VPC so Lambda functions in a VPC can call other Lambda functions, and (2) how to access S3 from a Lambda function in a VPC.

Profiling asynchronous functions on AWS Lambda is performed by pushing SAAF JSON objects to an S3 bucket for later retrieval using SAAF's **s3pull.py** script.

1. Download Project Source

To begin, using git, clone the project SAAF message passing example project:

```
git clone https://github.com/wlloyduw/saaf_mp
```

This project will open directly in Netbeans 11.

The message passing example project is a Lambda function that calls other Lambda functions for the purpose of spreading a message to a set of function instances.

The idea is derived from message passing protocols such as flooding and gossiping in TCSS 558 Applied Distributed Computing. With these protocols, we are interested in how many messages must be sent between nodes in unstructured adhoc networks to fully disseminate data. With AWS Lambda, each function instance can be thought of as a node. Nodes survive at least 5 minutes in the "warm" state before AWS Lambda selectively starts to remove nodes that have not been used for awhile as described by the paper Performance evaluation of heterogeneous cloud functions:

LINK:
http://faculty.washington.edu/wlloyd/courses/tcss562/papers/Fall2019/Group1-PerformanceEvaluationofHeterogeneousCloudFunctions.pdf

Our goal is to SPREAD a data message to every node in the network as efficiently as possible.

Unfortunately, in AWS Lambda, it is not possible for “nodes” to directly communicate. The only way Lambda nodes can exchange data (without using S3 for example) is by having a Lambda function call itself. When this happens, however, the function call is routed to a random node. There is no ability to control where requests will go.

The goal of the message passing example application then is to see how quickly (in seconds) a data message can be duplicated across all nodes for an AWS Lambda function. In addition to the time, we also want to make as few Lambda calls as possible.

The client only makes **one** Lambda call. This initial Lambda begins a process of message passing. Message passing occurs in rounds. In each round, the current Lambda function passes the message to **n** other Lambdas. When a Lambda function receives the message that already has it, then this node does not pass the message further. Only nodes newly receiving the message will pass the message on. The spread number indicates how many Lambda function calls each node newly receiving the message will attempt to make during each round.

Limitations: In practice, this application suffers from a couple challenges. First, the current implementation performs message passing sequentially, calling one Lambda at a time. Refactoring to call multiple Lambdas in parallel with multiple threads in Java will increase the efficiency of message passing. But even when this is done, Lambda only has 2 vCPUs. Therefore, the ability to send multiple messages at a time is limited. To have effective broadcasting, ideally we could send **n** messages in parallel (*or at least 10 would be nice...*). The second challenge is in how reserved concurrency works in AWS Lambda. When concurrency for a function is limited to lets say 10, the 11th call always fails generating the error: ***An error occurred (TooManyRequestsException) when calling the Invoke operation (reached max retries: 4): Rate Exceeded.*** Refactoring the application to trap this error and gracefully retry calls would be an improvement.

The message passing application provides an example application for asynchronous profiling in SAAF, as the client only makes one Lambda call, and there are many hidden Lambda calls in the background.

First, there are some modifications to make to the github code.

In the source code, navigate to: java_template/src/main/java/lambda/MessagePass.java

For this tutorial, it is recommended to use the **us-east-1 Virginia region**.

On **line 39**, update the **bucketname** variable to be an S3 bucket in the same Region as your Lambda function. If you don’t have an S3 bucket in US-EAST-1, please create a bucket.

On **line 166**, the **functionName** is hard coded as “tutorial9”.
If you deploy the function ***using another name***, the code will need to be updated here. You’ll also need to update the file message_passing/parfunction, and the message_passing/messagepass.sh script.

Please note example code:

Starting on **line 131** is an example of publishing SAAF data to an S3 bucket.

Starting on **line 74**, and combined with the interface defined on **line 164** shows how a Lambda function can be invoked by another Lambda function.

Please note:
A Lambda function in a VPC can't easily invoke itself. Putting a Lambda function in a VPC disables its Internet connection. The default behavior is for the Lambda function to **timeout** as Lambda functions can't be reached without the Internet. The workaround is to add a "NAT Gateway" to the VPC and set up the proper route tables. This is described as an optional activity at the end of the tutorial.

Additionally a Lambda function in a VPC, can't directly publish data to S3. There is an easier workaround however. A VPC endpoint can be added. Please contact the instructor with questions.

2. Build and deploy the message passing function

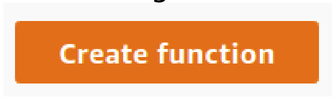
Using a favorite Java IDE or the command line, perform a maven build.

The next step is to deploy to AWS Lambda.

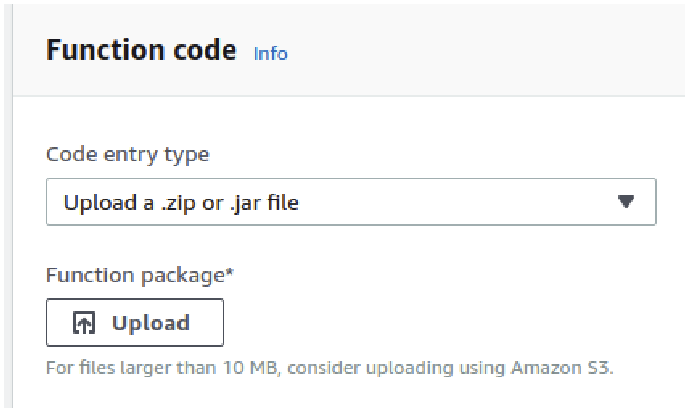
Navigate to "Lambda", and create a new function.

Using the wizard, use the "Author from scratch" mode.
Specify "tutorial9" as the function name, and "Java 8".
Selecting the defaults will create a new role for the function.

Once filling the form, click the button:



Next, upload your compiled Java JAR file to AWS Lambda:


Function code Info

Code entry type

Upload a .zip or .jar file

Function package*

Upload

For files larger than 10 MB, consider uploading using Amazon S3.

Click the “Upload” button to navigate and locate your JAR file. The jar file is under the “java_template/target” directory. It should be called “lambda_test-1.0-SNAPSHOT.jar”.

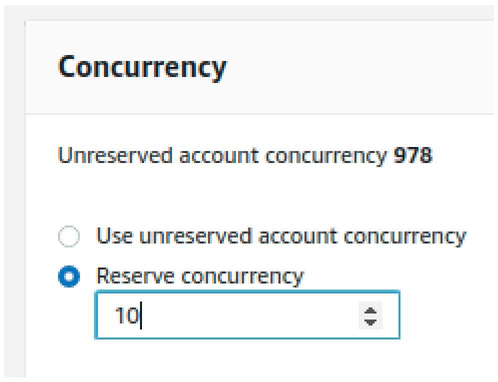
Once selecting the file, change the “Handler” to:

```
lambda.MessagePass::handleRequest
```

***** IF THE HANDLER IS NOT UPDATED, LAMBDA WILL NOT BE ABLE TO LOCATE THE ENTRY POINT TO YOUR CODE. THE LAMBDA FUNCTION WILL FAIL TO RUN *****

Next, for the message passing application, we would like to set the number of “nodes”. By setting the “reserve concurrency” setting on Lambda, it limits a function to a maximum number of function instances (e.g. like nodes).

Set this limit to **10.**



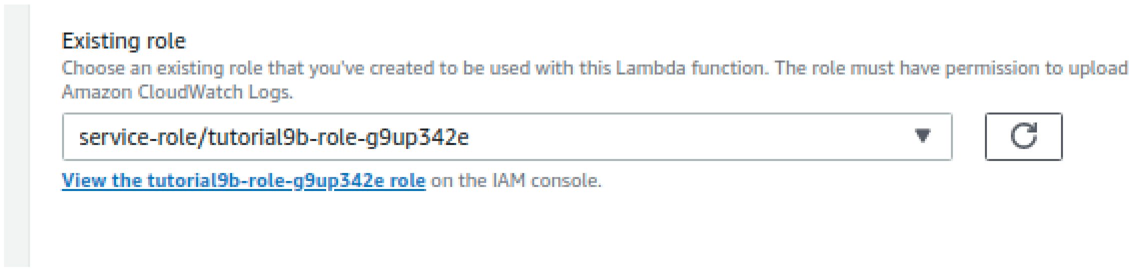
Unreserved account concurrency **978**

Use unreserved account concurrency [✓ Reserve concurrency]

Reserve concurrency: 10

Next, we need to modify the role to provide sufficient security permissions:

Scroll up to **Execution role**, and click on the “View” link to view the role in the Identity Access Manager:

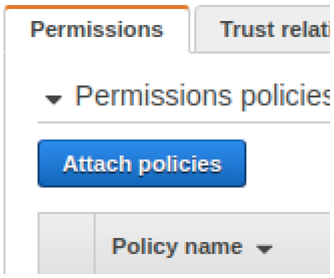

Existing role

Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload Amazon CloudWatch Logs.

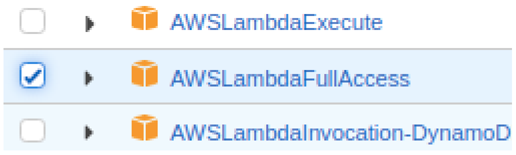
service-role/tutorial9b-role-g9up342e

View the tutorial9b-role-g9up342e role on the IAM console.

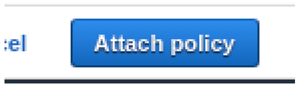
While viewing the role, let’s attach the “everything” policy (not very secure):



Click on “Attach policies”.
Then filter policies by “AWSLambda” and select “AWSLambdaFullAccess”:



Click the button to attach the policy to the role:



5. Testing the Application

To test the application, navigate to the message_passing directory where there are test scripts.

Perform the following sequence.

First initialize 10 nodes:

```
./init.sh 10
```

Then try message passing:

```
./messagepass.sh
```

Usage ./messagepass.sh {# of rounds} {node spread per round}

The first argument is the number of rounds, and the second is the number of messages to send per round.

Try:

```
./messagepass.sh 9 9
```

Here, making 1 Lambda call initiates 9 rounds of 9 calls each.

Then, check whether all nodes have received the message:

```
./check.sh 10
```

```
Fri Dec  6 01:30:27 PST 2019
Checking data spread nodecount=10 runsperthread=1 threads=10 totalruns=10

uuid,host,data,calls,totalcalls,uses,avgruntime_cont,avgsstuntime_cont,avglatency_cont
"1d7b3abc-01c6-440a-b044-8f9b1e752f4a",1575621251,"The Data",10,0,1,14478,12160,2318
"1b3df4d8-7664-4c36-84ca-f5163ecae9bc",1575621182,"The Data",16,0,1,14540,12175,2365
"92fdbae6-f8f4-4b6c-ad66-a97a2e98ccfc",1575622149,"The Data",1,0,1,14566,12267,2299
"32b4c15c-08fa-4855-ab30-c9e470bcaf7f",1575621126,"The Data",1,0,1,14587,12274,2313
"733a422b-f148-4b97-bbc6-a3331238dc91",1575623281,"The Data",1,0,1,14610,12209,2401
"a3926221-b3ed-40f5-ae4e-3e8287fd48e2",1575623154,"The Data",3,0,1,14589,12212,2377
"4e20075b-daf3-4d2c-a255-e2d788aa1d04",1575622027,"The Data",12,0,1,14881,12100,2781
"d0065991-d779-4039-a201-4bdcdeaf2265",1575622873,"The Data",1,0,1,16383,12196,4187
"ad7fb301-43ef-4ced-a9bf-b05613dca76f",1575621186,"The Data",15,0,1,16960,12361,4599
"dc6e2330-10f9-4d16-93ae-e2ad1318ca63",1575621690,"The Data",1,0,1,17527,12281,5246
Current time of test=1575624646
```

Here, we've been able to spread the message "The Data" to all 10 nodes without using S3 by making only one Lambda call. This was very inefficient however, as many Lambda calls were required.

The number after "The Data" indicates how many calls each node received.
Lambda does a poor job at distributing the calls. (10, 16, 1, 1, 1, 3, 12, 1, 15, 1)

6. Profiling with SAAF

All Lambda calls have been publishing JSON results to an S3 bucket.

Using the s3pull.py script, SAAF can grab and analyze results.

Navigate to the test directory:

```
cd ../test
```

```
./s3pull.py tcss562.mylogs.aaa experiments/mp.json 0
```

** ***Replace tcss562.mylogs.aaa with the name of your bucket.*** **

The s3pull script pulls all files from the S3 bucket, and produces a CSV report file. If your platform automatically opens CSV files, it will immediately appear in a spreadsheet. If not, using a spreadsheet application such as Microsoft Excel, open the CSV file.

CSV files are saved locally under the history directory.

<u>SUBMISSION</u>

To submit Tutorial #9, upload the CSV file produced by s3pull.py to Canvas.

Scoring

20 points	Providing a CSV file showing the message passing application running the commands: init 10, messagepassing 9 9, check 10

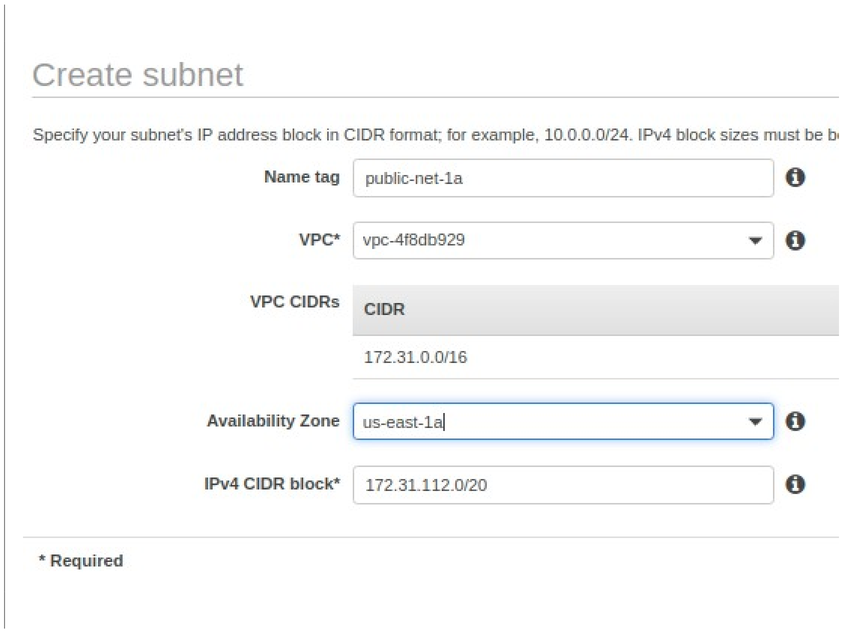
Tutorial 9 - Optional Activity

Configuring a NAT Gateway to provide Internet access to Lambda Functions running in a VPC

To enable a Lambda function in a VPC to call a Lambda function in a VPC or not, it is necessary to add a "NAT Gateway" appliance to the VPC and add route tables. A "NAT Gateway" acts like a network router device. It routes traffice from the private subnet through the NAT Gateway and out through the public subnet. The major drawback is that NAT gateways cost 4.5 cents/hour, or $1.08/day, $7.56/week, $32.40/month, or $394.20 a year. Because of this cost, NAT gateways should be set up for temporary usage only.

To begin, navigate to "VPC". For the VPC, let's create two subnets, one public, and one private. First, we need to identify your account's **default** VPC. Click on "Default VPC". In the display, look for the row with Default VPC labeled as "Yes". Note the VPC ID.

Next, on the left side-bar select "Subnets". And then click "Create Subnet":


Create subnet

Specify your subnet's IP address block in CIDR format; for example, 10.0.0.0/24. IPv4 block sizes must be b

Name tag	public-net-1a
VPC*	vpc-4f8db929
VPC CIDRs	CIDR: 172.31.0.0/16
Availability Zone	us-east-1a
IPv4 CIDR block*	172.31.112.0/20

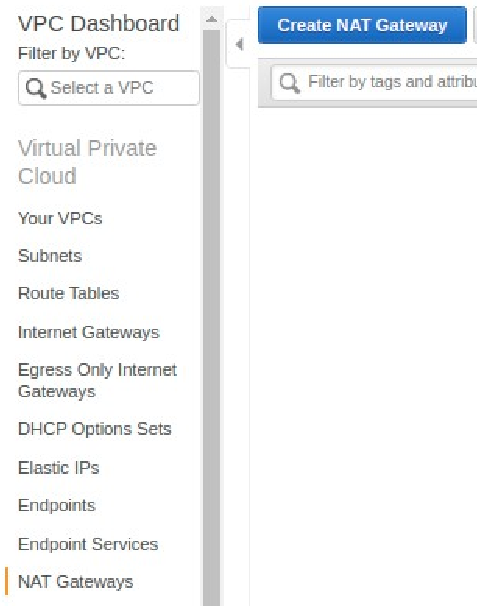
* Required

Specify a name: public-net-1a (where 1a is the availability zone)
Select the default VPC discovered above.
Choose an availability zone, for example in Virginia us-east-1a (or 1b, 1c, etc as desired).
Then specify a IPv4 CIDR block. This block must not already be defined by another subnet in the VPC.
Try: **172.31.112.0/20**
Then click **Create.**

Now, create a second subnet.
Specify a name: private-net-1a (where 1a is the availability zone)
Select the default VPC discovered above.
Choose an availability zone, for example in Virginia us-east-1a (or 1b, 1c, etc as desired).

Then specify a IPv4 CIDR block. This block must not already be defined by another subnet in the VPC.
Try: **172.31.128.0/20**
Then click **Create.**

After creating both subnets, next on the left side-bar select “NAT Gateways”.
The click “Create NAT Gateway”:



In the user interface,
Subnet: select the subnet ID for the public subnet you just created
Elastic IP Allocation ID: Click the button for “Create New EIP”.

PLEASE NOTE, CREATING AND SAVING AN <u>UNUSED</u> ELASTIC IP COSTS $3.60 a month. Elastic IPs are FREE when they are actively associated with a NAT Gateway or an EC2 instance. When they sit in your account unused, they cost 12 cents a day or $3.60/month ($43.20/year).

When ready, create the NAT Gateway:

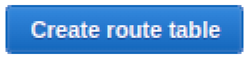
Remember to DELETE the NAT Gateway after this tutorial as the cost is $1.08/day or $32.40/month to retain it in your account.

Once the NAT gateway has been created, click on the “Internet Gateways” link in the LEFT-Hand sidebar menu:

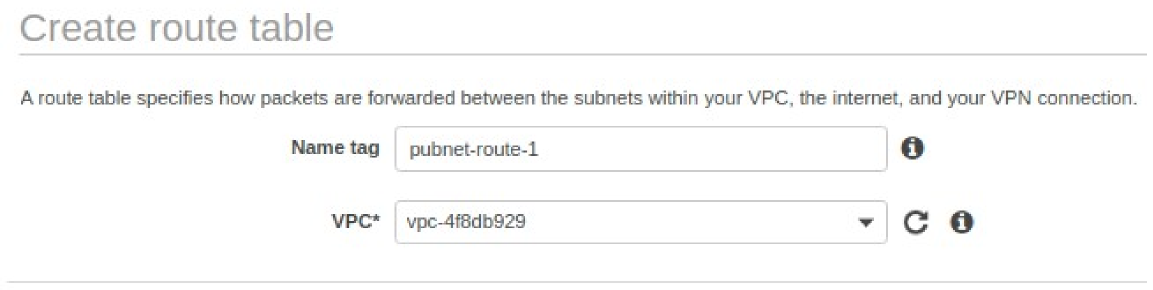
Identify the ID of the Internet Gateway that is attached to your Default VPC.

Next click on the “Route Tables” link in the LEFT-Hand sidebar menu.

Create two new route tables, one for the public subnet, one for the private subnet:



Provide a name such as “pubnet-route-1” and select the default VPC:



Create route table

A route table specifies how packets are forwarded between the subnets within your VPC, the internet, and your VPN connection.

Name tag: pubnet-route-1

VPC*: vpc-4f8db929

Repeat the procedure creating the private network route:
Name tag: privnet-route-1
VPC: (your default VPC ID)

Now, select the pubnet-route-1, and click on the “Routes” tab, and select the “**Edit Routes”** button.

Click the “**Add Route”** button.
Destination: 0.0.0.0/0
Target: (select “Internet Gateway”, then the Internet gateway ID from above)

Then click the “**Save routes**” button.

Then click on the “Subnet associations” tab.
Click the “**Edit subnet associations”** button.
Identify and select your public subnet (public-net-1a).
Click Save.

Next, select the “privnet-route-1” and select the “**Edits Routes**” button.
Click the “**Add Route”** button.
Destination: 0.0.0.0/0
Target: (select “NAT Gateway”, then the NAT gateway ID you’ve created from above)

Then click the “**Save routes**” button.

Then click on the “Subnet associations” tab.
Click the “**Edit subnet associations”** button.
Identify and select your private subnet (private-net-1a).
Click Save.

Now, while still in VPC, create an “Endpoint” so the VPC can access S3.
Click on the “Endpoints” link in the LEFT-Hand sidebar menu.

Click the button “**Create Endpoint**”.
Scroll down through the long list and select: “com.amazonaws.us-east-1.s3”
Make sure the default VPC is selected.
When the route tables, appear, select your public and private route tables.
Then create the endpoint, by clicking the button:

Next, to use the VPC with the newly configured NAT Gateway and routes, go to your Lambda function, and select the VPC, and private subnet (privnet-route-1) you’ve just created. You’ll also need to select a security group. The “default” security group should work.

It should now be possible to run the message passing application using the Lambda function in a VPC. The Lambda should be able to call itself, and write to S3.

```
./init.sh 10

./messagepass.sh 9 9

./check.sh 10

cd ../test

./s3pull.py tcss562.mylogs.aaa experiments/mp.json 0
```

***** Replace tcss562.mylogs.aaa with the name of your bucket. *****

For additional information, see:

https://stackoverflow.com/questions/39144688/aws-lambda-invoke-not-calling-another-lambda-function-node-js

https://aws.amazon.com/blogs/aws/new-access-resources-in-a-vpc-from-your-lambda-functions/

Be sure to DELETE the NAT Gateway and ALL Elastic IP addresses in the VPC after completing this tutorial as the cost is $1.08/day or $32.40/month to retain the NAT Gateway, and 12 cents/day, $3.60/month, or $43.20/year to retain each Elastic IP address in your account.