CREDENCE

USER GUIDE

License

Information

This guide provides information about using the CREDENCE distributed platform and some supporting information. If you want to contact the CREDENCE team, please go to the end of the guide for more information

Table of Contents

Overview

What does CREDENCE stand for?

CREDENCE is short for Cloud based Reliable Distributed platform for Information Coding Experiments. Although this project was started as a platform for distributed coded computation algorithm prototyping and testing, we believe it would be ideal for any distributed computation algorithm that fits the Master Worker Framework.

What does CREDENCE do?

CREDENCE provides an easy way to implement and test algorithms that fall under the Master Worker Distributed Computational Paradigm. It sets up listeners at Master and Worker Nodes that listen to queries and serve the respective API calls. It also abstracts away the tedious communication APIs and provides efficient Data Abstractions that are instrumental in speedy prototyping. Currently, the framework provides automation scripts to set up the master and worker applications on AWS Cloud using EC2 Instances.

CREDENCE also provides complete isolation between the deployment process and the application design. This makes the deployment of the application on any other cloud or native cluster platform easier.

Why is CREDENCE useful?

This project provides an extremely effective way to write distributed algorithms and test them on real-world systems. It is a flexible framework for prototyping distributed algorithms. It provides automation scripts that help developers to securely deploy (and subsequently destroy the infrastructure after experiments) their distributed algorithm on top of a massive deployment of worker nodes in AWS Cloud with the flexibility to choose different machine types and relevant setup details. There are other automation scripts that reduce the efforts of running an experimental setup and lets researchers focus solely on algorithm development. With future extensions to standard Math and ML libraries (like MTJ-N and Spark MLLib), the project will be able to support all types of linear algebra operations and yet provide standard serialization and deserialization of objects, file reading and writing interfaces for these specialized math libraries, which would greatly simplify algorithm implementation.

What is the future of CREDENCE?

We have tons of features in development, suggested by researchers and developers like you. If you have ideas that you believe would help you gain more value out of CREDENCE and are not already listed in our Future Goals and Roadmap, then we would suggest submitting a feature request using our contribution guide.

What is CREDENCE built on?

We have used the following tools for building CREDENCE.
Maven, Java, Shell Script, Log4j2, Undertow, and Unirest.

All algorithm development currently must be in JAVA. We plan to provide support for other languages in the future.

Getting Started

Setup

We have tested our code on Ubuntu 16.04 and macOS High Sierra. We do believe though that without major modifications, CREDENCE could be run on any Linux or Mac OSx based machine.

Setup instructions for Ubuntu

sudo apt-get update
sudo apt-get install -y default-JDK
sudo apt-get install -y maven
sudo apt install python-pip
pip install awscli --upgrade --user
sudo apt-get install jq
git clone https://github.com/MalharSC/aws_master_worker_framework.git

If you are using another Linux distribution, you could modify the above commands to suitable package installation commands.

Setup instructions for macOS

brew cask install java
brew install maven
brew install python
pip install awscli --upgrade --user
brew install jq
git clone https://github.com/MalharSC/aws_master_worker_framework.git

Quick Startup Guide

1. Find a Linux or Mac OSx based machine.
We have tested our code on Ubuntu 16.04 and macOS High Sierra. We do believe though that without major modifications, Credence could be run on any Linux or Mac OSx based machine.
2. Clone our GitHub repository using the following command (Provide GitHub Link):
3. Check the examples in our repository to understand how algorithms have been implemented. The AppMaster.java and AppWorker.java file within the master_setup directory and worker_directory would help in understanding how to implement distributed algorithm.
4. Implement your algorithm. You should have to modify only the AppWorker.java, ServletWorker.java, AppMaster.java and ServletWorker.java files within the master_setup and worker_setup directories in the root to implement your algorithm. Log all your results using the Log4j2 logger within your code.
5. Run the deploy_infra.sh, script to deploy your code on AWS Cloud.
Trigger the experiment by sending an init curl request from your machine to the Master node.
6. After the experiment is over, run get_results.sh to download all the results from your experiments to your local machine.

7. Run the clean_infra.sh script to destroy all the infrastructure deployed on AWS Cloud.

You have implemented your distributed computation algorithm! For any clarifications, follow the detailed section-wise guide and FAQs described below.

Installing git-secrets

git-secrets is a utility that prevents you from committing keys and secret credentials into git repositories. Please follow the guide here https://github.com/awslabs/git-secrets to set up git-secrets for your repository.

Note on Terminology

In this guide, we will use instances, nodes, and machines interchangeably to identify a single independent computing entity.
The complete deployment process is set up on AWS Cloud and hence there are certain jargons which are borrowed from AWS documentation, and sufficient efforts have been taken to explain them in this guide. But if the user still finds difficulty in understanding them, browsing the AWS guides available on the internet would provide solutions and answers to these doubts.

Setting up an AWS Org

AWS Organizations is a service provided by AWS to set up a central parent organization under which you can connect multiple accounts and support consolidate billing. The Parent account gets billed for all the AWS resource usage by the Child account and can be used to centrally manage funds and monitor AWS usage. You can also use AWS Org to enforce policies on Child Accounts in a large group. For more information please refer to the link: https://aws.amazon.com/organizations/. Please follow the link here to set up and AWS Org: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_create.html.

DEPLOYMENT PROCEDURE

IAM User

Overview

In most scenarios, your default AWS account user is the root user, unless you have created a custom IAM user. As a root user account, you have access to create any resources on AWS without any limitations (other than those placed by AWS on all accounts). This also means that loss of Access ID and its corresponding Secret Access Key (used for programmatic access to AWS resources via CLI) could mean anyone would be able to use your account to start any resources. To avoid such a fraudulent scenario, we could use AWS Identity and Access Management (IAM) service to create a custom user with lesser privileges (whatever may be required for your experimental setup), and protect ourselves against accidental key loss and they damage occurring because of the key loss.

Creating a custom IAM user

We will create a custom user, with extremely limited access to resources within your account. Sign into your AWS Management Console and find the link to IAM within the Services Tab Click on Users (in the left navigation pane) and add a new user. Give the user a name, enable programmatic access (necessary) and management console access (optional).

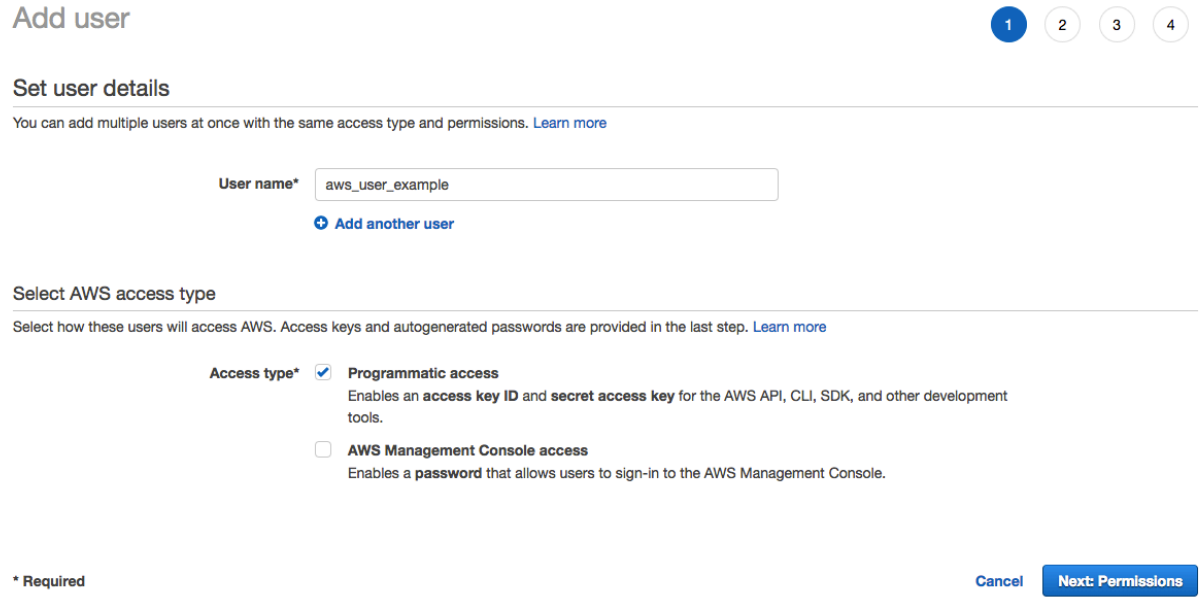

Fig 1: Add User Page within AWS IAM Console

Click Next and Click on Create Group. Provide a Group Name and click on create policy in the pop-up tab.

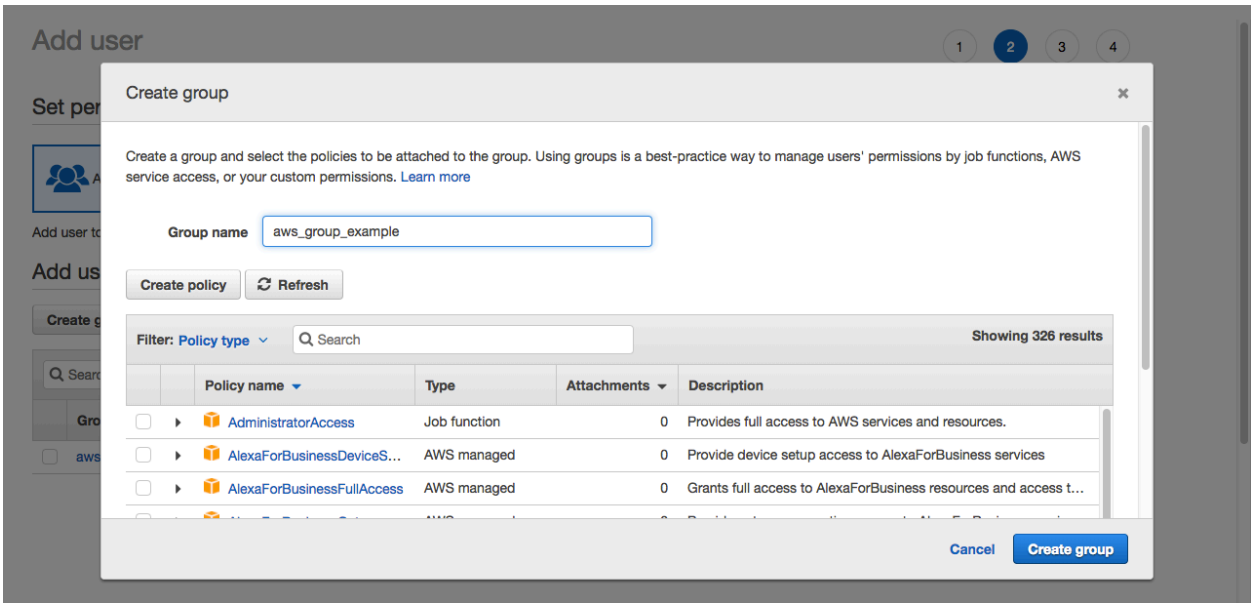

Fig 2: Create Group Pop Up on the second page of Add User Wizard

This will open a new screen and click on the JSON tab. Paste the sample policy at the end of this section. Click on Review Policy and if there are no errors, give a name to the policy, provide a Description and click on Create Policy.

Fig 3: Creating Policy limiting access to the Custom User

Return to the Create Group Page, and find the policy you created (click Refresh if necessary) and click on Create Group.

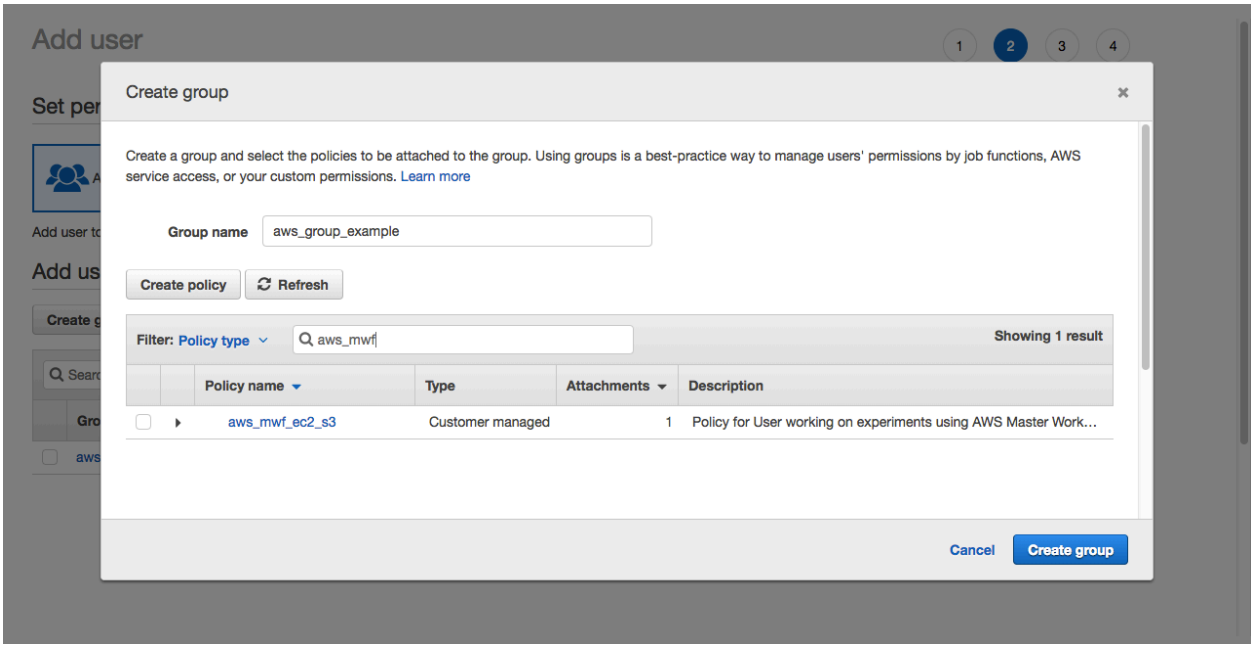

Fig 4: Selecting the created policy within the Create Group Tab

Click on Review and on the next page, click on Create User.
This page provides the Access Key ID and Secret Access Key for the User. Store it securely in your local space. They will be required for configuring the user later, in deployment step. If you lose the Secret Access Key, then the only way to run your experiments would be to delete these pair of credentials from your console and generate another pair of Access Key ID and Secret Access Key.

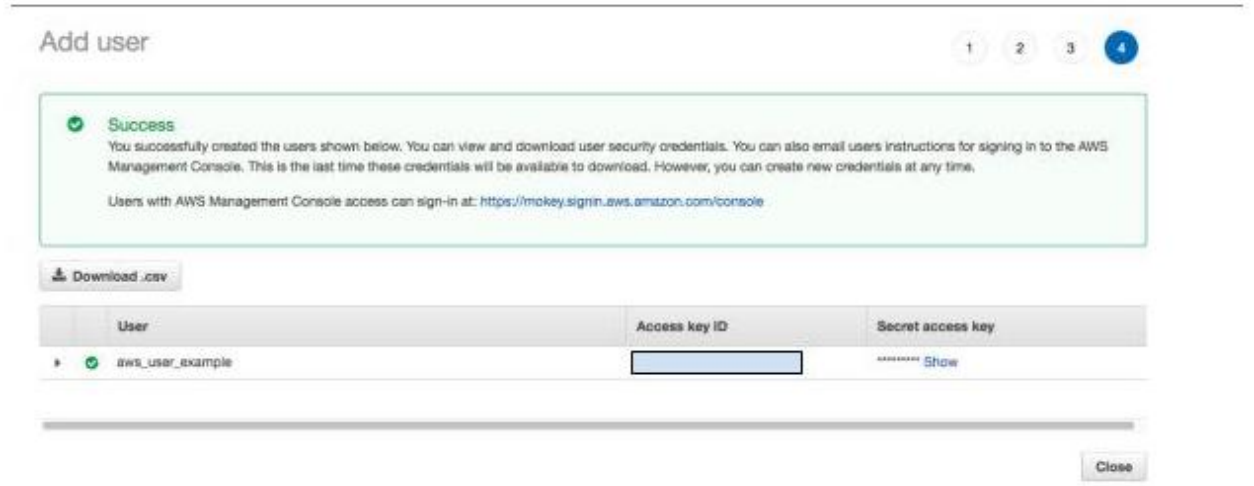

Fig 5: User Created with Programmatic Access

Deploying Infrastructure

Overview

The deployment script automates the task of deploying master and worker nodes on AWS Cloud. The script can be called by running ./deploy_infra.sh {deployment_function_name}. The brief description of the deployment process is as follows:

1. Initialize the AWS Command Line Interface. This step connects your IAM user credentials to your local machine. Please find the more detailed explanation below.
2. It creates IAM Role, IAM Role Policy, Security Group, and Public-Private Key Pair. These components are explained in more detail in the following sections.
3. The script asks for your inputs on the type of Master node that needs to be launched.
4. After creating the Master node, the script goes on to create a set of Worker nodes, based on the inputs provided by you.
5. Once all the nodes are running and have passed the status checks, the script initiates launching CodeDeploy agents on all the nodes. More details on CodeDeploy agent follows in the description below.
6. Next, the script creates S3 buckets to which the corresponding Master and Worker applications are packaged and stored.
7. In the final step, the script triggers the CodeDeploy agents to gather their respective applications from the S3 buckets and deploy them on the nodes in parallel. The CodeDeploy agent has access to a configuration file, present within the application, which it uses to orchestrate the setup. More information on this step is available in the description below.

The deployment process was designed to be able to run with failures in different stages of the process. If the deployment fails at any given stage, the deployment script tries to exit the infrastructure to allow the user to identify the failure and take the necessary steps to resolve them. The manual intervention steps that could be taken are usually written in the error messages. Once the failed deployment stage is fixed, one can re-run the deployment script with arguments for the steps that follow the failed steps individually. All the steps in the deployment process are completely variable independent and hence can be run without any inter-dependencies even after failure.

Demo Deployment

The following steps outline the step by step process for initiating a standard deployment process in CREDENCE. The details about the different components being used within this deployment script are provided in the next section.

Run the deployment script '**./deploy_infra.sh all**' from within the deployment scripts folder. The '**all**' argument ensures that all the steps required for deployment are run sequentially.

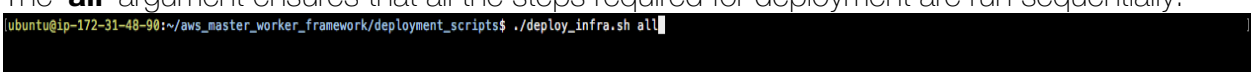

Fig 6: Starting the Deployment Process

The deployment script prints the information on all the sub-commands that can be run as a part of the deployment process. In case of pushing just updates or failure scenarios, you can choose to run the individual steps separately. The first input to the deployment process is the AWS CLI (Command Line Interface) Configuration, which requires the IAM user Access Key ID, Secret Access Key and the Default Region of Operation for this CLI user.

Fig 7: Setting up AWS CLI during Deployment

As soon as you provide the IAM credentials for configuring AWS CLI, the deployment script creates an IAM Role and Instance Profile with appropriate Role Policies attached to it. This IAM Role is different from the IAM user used to configure the CLI in the previous step. The deployment script also creates a Security Group that lets the instances be accessible from anywhere on the internet using any protocol. It also creates a Public-Private Encryption Key Pair that is used for providing authentication and access to the instances.

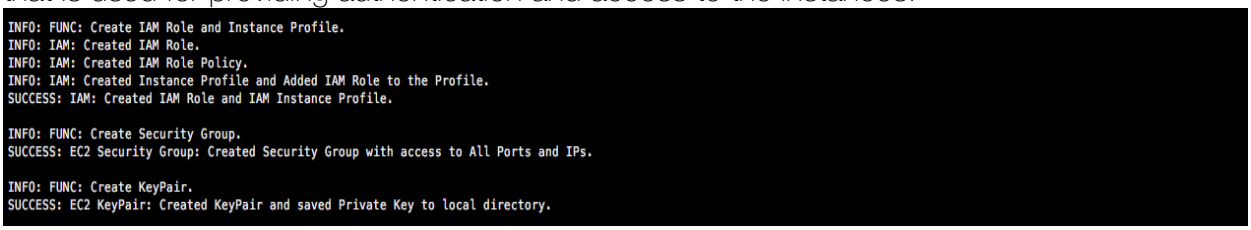

Fig 8: Creating IAM Role, Security Group, and Key-Pair

Next, the deployment script asks for details about the Master node to be launched. Following are the inputs that are asked during the Master node creation process:

Master Node Type: This is the AWS Instance Type that the Master node should be launched on. The default option for this input is **'t2.small'**.

Master Node Tagging: The deployment script also asks for a Key-Value pair for custom tagging, and becomes extremely helpful in keeping tabs on the AWS usage cost for different projects within the same organization. There is no default input available for this option, thus, in case if you leave this input empty, it creates an instance with empty Key-Value pair tag.

Master Node Volume Size: Next, you must provide the Volume size for the Master node. The default option for this input is 8GB, which is also the default Volume size with which AWS launches its t2.small instances.

Master Node Availability Zone: Within a given region (like us-east-1) you can choose to launch the instance in different Availability Zones (like us-east-1b or us-east-1a). The reason for following this approach could be to either launch instances in multiple Availability Zones or to

reduce cost owing to the availability of cheaper spot instances in one Availability Zone compared to other.

Master Node Instance Launch Type: The user can choose to launch the instance as a Spot Instance or an On-Demand Instance. While the Spot Instance might be a lucrative option to reduce the actual dollar cost of the experiment, it might not be the most reliable option. Please read the next section for more details.

The deployment script based on the inputs provided launches the Master node and waits for the Master to successfully pass all initialization checks. It also creates an additional tag '**Type:aws_mwf_ec2_master**' on the Master instance for the purposes of identifying Master and Worker nodes separately**.** If the tag creation process fails, the later part of the deployment will run into a cascade of failures. To avoid this failure, check the deployment process debugging guide below.

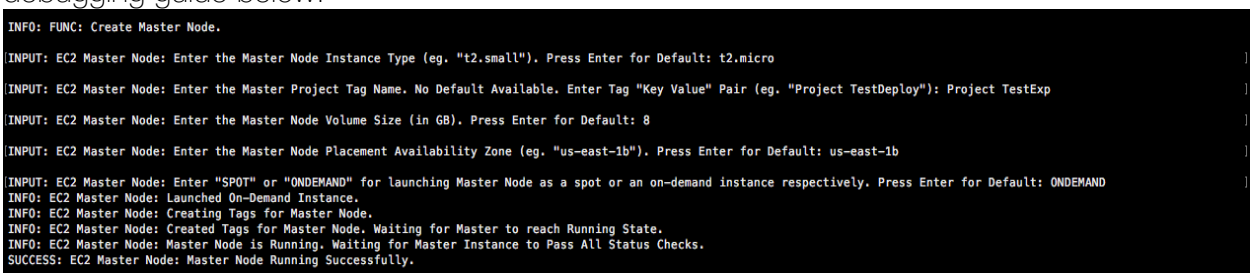

Fig 9: Creating Master Node

After the Master node is created, the deployment script starts creating the Worker nodes. The inputs to the script are the same inputs that were asked while creating the Master node. Additionally, the script asks for the number of Worker nodes to be launched. Although it is strongly recommended to not leave this option blank, the script assumes 1 worker by default and proceeds the deployment setup. Additionally, for the Workers, the deployment script also prints out the Instance-IDs as they are created for helping with easy debugging in case of deployment failures.

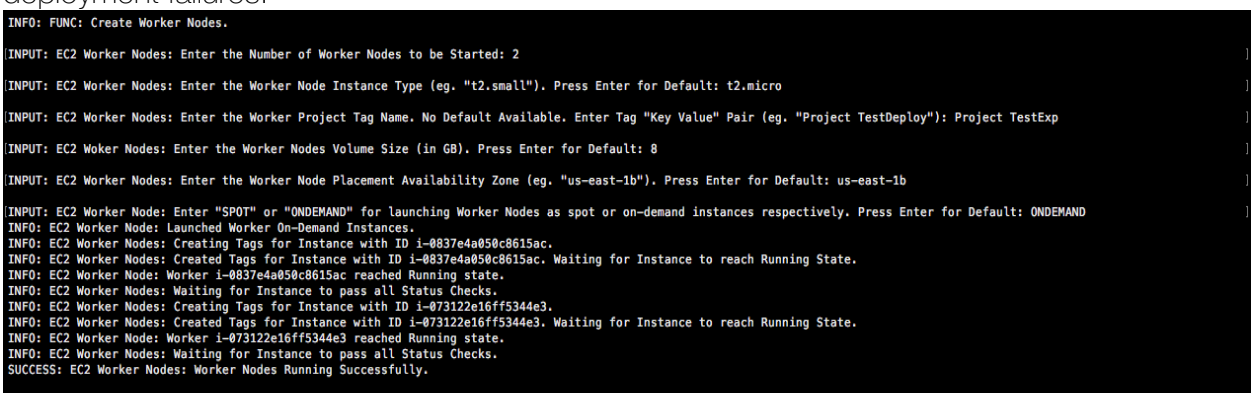

Fig 9: Creating Worker Nodes

Once the Master and Worker nodes have been launched, the script proceeds with the installation of CodeDeploy agent on all the instances. The setup takes multiple attempts at installing the agent, failing which it may exit the infrastructure setup, requiring either running this step of the deployment process again or manual intervention. The CodeDeploy agent

orchestrates the setup of Master and Worker servers on all the instances including the installation of dependencies for these servers. For more information on CodeDeploy, please refer to the next section.

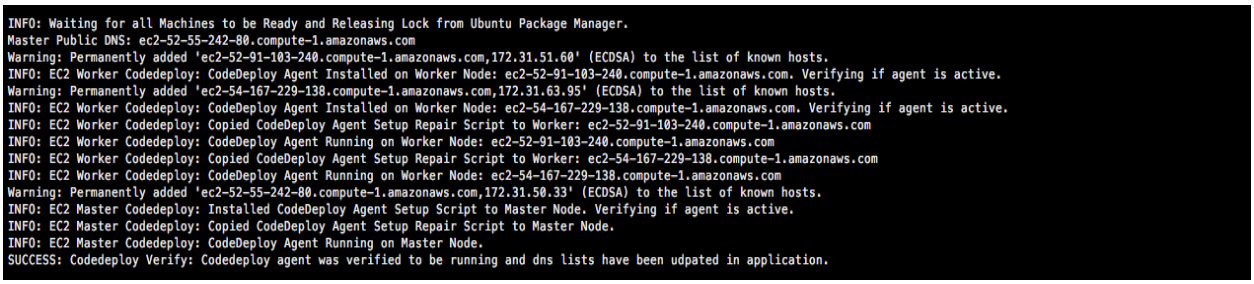

Fig 11: Installing CodeDeploy Agent on all Nodes

Once the CodeDeploy agents are installed and are actively running on all the instances, the deployment script creates two S3 Buckets. One each for loading the Master and Worker application. Launching an application via an S3 bucket provides a standardized way for CodeDeploy to launch the applications on all instances at once, without being dependent on your local network bandwidth or machine state for uploading application files.

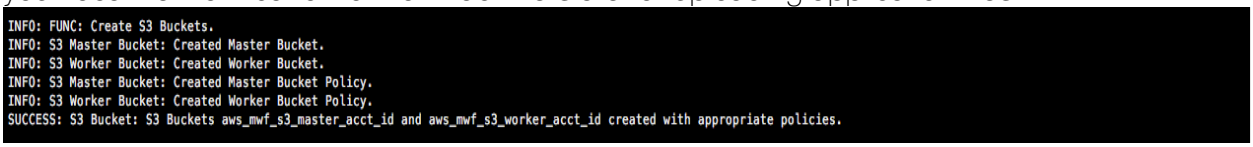

Fig 12: Creating S3 Buckets for uploading Application

Once the S3 Buckets are created, the deployment script packages the application code and uploads it to the corresponding S3 bucket. Then with the help of CodeDeploy agent, it initiates '**AllAtOnce**' deployments on the Master and Worker instances and waits for successful deployment on all instances. Although not necessary, if your application requires changes to the dependencies or any part of the application deployment lifecycle, you can refer to the next section on how to modify these defaults. There might be times when the CodeDeploy deployment might fail, especially on the Worker nodes and you can refer to the debugging section below for more help.

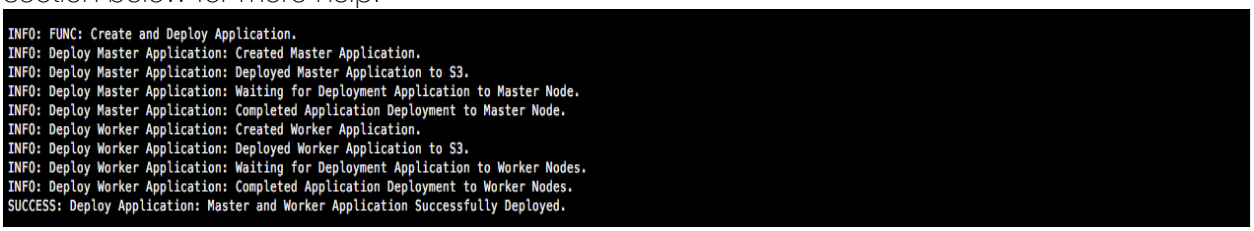

Fig 13: Creating and Deploying Application on all the Nodes

Once the Master and Worker nodes are deployed, the Master node needs to be triggered to start its computation. This is usually achieved by sending the master an INIT request, of the following type. The request to be sent would largely depend on how this INIT is implemented within the application and more information can be found in the Application Description section.

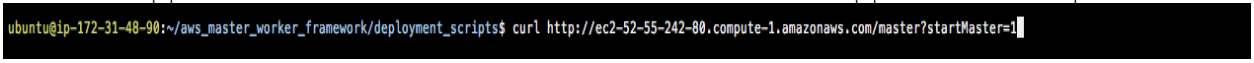

Fig 14: Sending a trigger to Master Node for initiating experiments

Once the Master is triggered, it will continue to run the experiment until completion. The results or traces from the experiments are logged using the Apache Log4j framework. More information on logging is available in the Application Description section. The logger stores the results from the applications locally in their respective results directory. The following script, get_results.sh collects the results or the logged files from all the nodes and downloads them to your machine at the parent folder. It creates a folder mentioning the timestamp at which the results were downloaded. The results are further kept in different subdirectories identified by the EC2 public DNS identifier of the node from which they were downloaded.

Fig 15: Download Results to User Machine from all the Nodes

After downloading all the results, and verifying that the experiment ran successfully, you can initiate infrastructure tear down by calling the **'clean_infra.sh all'** command. This script takes care of terminating all the instances that have been started in the deployment process. It also deletes the Security Group, IAM Roles, KeyPair files, S3 Buckets and the CodeDeploy application started as a part of the deployment process. Failure in this step usually rises from either the script trying to delete a resource which is already deleted or some runtime error due to resources still being in use. In case of errors please follow the Debug guide, which provides some ideas for debugging the errors if any during this step.

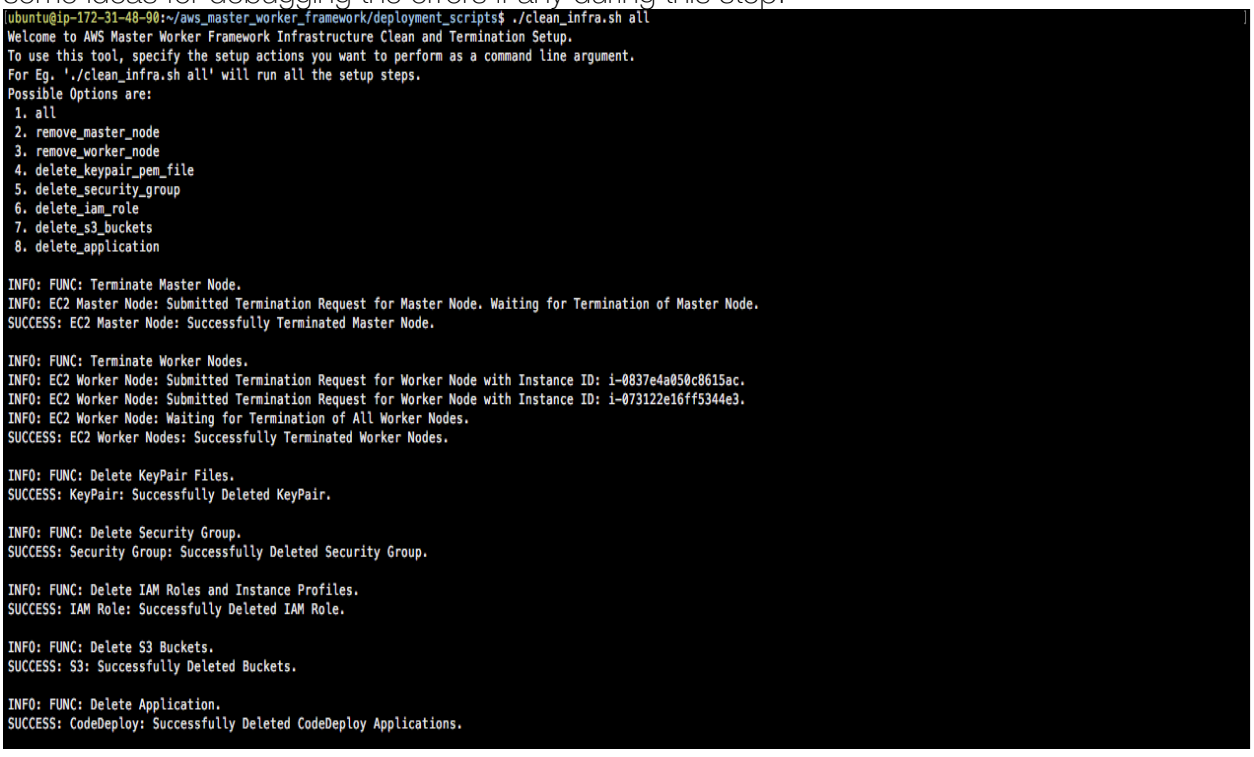

Fig 16: Deleting Deployed Infrastructure

This completes the Deployment Process Demo. For more information about the individual resources explained in the demo, details on Error Codes used in the deployment process or general debugging guide, please go through the following sections.

For possible options in the deployment process, run the deployment or clean infrastructure script with the argument **'help'** and it will print out all the possible argument options to the script.

Component Overview

AWS Access ID, Secret Access Key, and Default Region

For programmatically accessing AWS resources (that is creating AWS Resources without using the web console), the AWS SDK or CLI requires authenticating the user using some access credentials. When you create an IAM user with limited access rights than the root user, the IAM wizard generates these Access Credentials (Access ID and Secret Access Key) for you (but only if you choose programmatic access option within the wizard). You must store these credentials locally in a secure folder to avoid misuse. It must also be noted that you can access these credentials only once when they are created and there is no way to retrieve them later, other than to create another pair of access credentials. For more details about access credentials please go through the AWS documentation available here: https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html#access-keys-and-secret-access-keys.

IAM Role, Security Group, and Public Private Key Pair

IAM Role is a set of policies that define access to different AWS resources and can be attached to an instance via an Instance Profile. The IAM Role is different from an IAM user in the sense that although there are policies attached to the IAM Role, the Role itself doesn't have any long-term access credentials like the user. Security Group defines the network access rights of an instance and defines who can connect to the instance from an external domain. A Public-Private encryption key is used to provide secure access to your instance. For more information about IAM Role and Instance Profile follow this link: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html and https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2_instance-profiles.html. Further details about Security Group and Public-Private Key Pair can be found here: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html and https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html.

CodeDeploy Agent

CodeDeploy is an AWS Service that provides support for deploying applications on EC2 instances. For this service to operate, it requires agents (aka service) running on your EC2 instances. These agents speak to the AWS CodeDeploy service which orchestrates the complete application deployment process. This orchestration includes installing dependencies, building the application and executing the build. It also provides a safe way to kill the running application and push application revisions to the instances. The Application Specification Configuration file which defines these Application Life Cycles is available in both the Master and Worker application folders under the name appspec.yml. The Application Specification Configuration file, in turn, uses scripts available in the scripts/ directory of each application folder

to execute the various Application LifeCycle event. These scripts can be modified to suit the application requirements.
For more information about CodeDeploy please follow the link: https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html and for information about how to use CodeDeploy with EC2: https://docs.aws.amazon.com/codedeploy/latest/userguide/instances-ec2-configure.html.
The deployment script takes reasonable measures to install the CodeDeploy agent. It tries multiple attempts at launching the agent in case of failures the first time. If the agent installation continues to fail, follow the below-described actions:
Choose a Worker or Master node and SSH into it using the AWS Key-Pair.
Check if the CodeDeploy agent is active or disabled by running the following command:
sudo service codedeploy-agent status.
For more information on how to verify if the CodeDeploy agent is running or not can be found here: https://docs.aws.amazon.com/codedeploy/latest/userguide/codedeploy-agent-operations-verify.html.
The script instance_setup_ubuntu_repair.sh should be available at the home directory **'/home/ubuntu'**. Try running this script **i**n case of CodeDeploy agent installation failure, and observe the error codes returned by it to resolve any installation errors.

S3 Bucket

If the buckets are existing already, when the **'create_s3_bucket_update_policy'** step of the deployment script is run, then the script seems to be stuck and takes a long time before returning an error code.
Usually deleting the existing buckets and running this part of the deployment resolves the error.
Also, sometimes, you might observe failure when the deployment script tries to run a get command to verify if the policy created on S3 bucket exists. You can verify whether the policy exists on the S3 buckets by visiting the AWS S3 Web Console and if it does exist, then you can ignore this message and proceed to the next part of the deployment.

Codedeploy

The final part of the deployment process usually doesn't fail, unless some of the following scenarios are encountered:
1. If the objects master_setup_app.zip or worker_setup_app.zip are present in either the Master or Worker buckets on S3 before the script was executed. The deployment script will be stuck a long time, and would eventually fail. One can resolve this issue by deleting these objects from the S3 buckets using the AWS S3 Web Console.
2. The deployment process would also fail if a CodeDeploy Application is existing from the previous run. If you plan to push a revision rather than initiating a fresh deployment, then you should simply run the script with the push_updated_application argument.
3. In some cases after creating a CodeDeploy deployment, you might encounter a **'Waiter Terminal Failure'**. This usually points to either error with the CodeDeploy Application Specification file (appspec.yml) or some of the installation scripts available under the /scripts directory of either the Master or Worker application. To view the logs of the failure, follow the steps described below to understand the root cause of failures.

In certain rare cases, the deployment successfully proceeds but you receive multiple connections reset errors when either connecting to the Master or when you see the error logs for the requests sent from the Master to the Worker nodes. If the server was set up correctly then this is most probably, the case when the server has failed to launch on the nodes. In such scenarios, SSH into the Master or Worker nodes and try to run the following command **'ps aux I grep java'.** If the command returns no message, then probably the server was not started at all and trying to build the application using Maven would show the build errors. More information on application debugging is present below in the Application Debug section.

Pushing an updated application
If you want to update the application for either error in the code or changing some parameters, you can do so by calling the deploy_infra.sh script with the argument '**push_updated_application**'. This option will allow you to update either the Master, Worker or Both the applications based on your input**. T**he updated application will be packed and uploaded to S3 from where CodeDeploy will deploy it to the specified instances. It will safely kill the running application and then proceeds to install the new application, based on the scripts defined by the developer.

Debug Guide

Overview

The most important debugging tool for Deployment is the AWS Web Console which can be an excellent way to find whether a EC2 resource was created or not and to view the failure logs.

If any step pertaining to AWS IAM Role, Security Group and Key Value Pair creation or deletion fails, you can go the AWS EC2 Web Console and check whether the resources were created with the right properties. In case of failure during the cleaning infrastructure step, you can go to the AWS EC2 Web Console and delete the resources manually.
Insert Images of the console for all the three types.

If the deployment script fails during Master or Worker Node creation, you can go to the AWS EC2 Web Console and verify if the script created any instances. This part of the process can have failures for multiple reasons. Some of them are described below:

Error Code Explanation

AWS Dry-Run Operation: The deployment script executes a dry-run operation, before actually calling the command to create either Master or Worker instances. These dry-run operations provide an excellent opportunity to explore any problems with the inputs before running the actual create instance command.
Errors that don't begin with **'ERROR'** title are usually errors thrown by AWS CLI and information about them could be found within the AWS Documentation available on the internet.

For the errors thrown by the deployment script, they start with the label '**ERROR**' and usually define the operation that failed. The error message also provides some possible ways of debugging the failure and fixing it. The steps to fix either point the user to the console for finding out the actual cause of the error or would suggest cleaning a resource and trying out the failed deployment step again.

Script Stuck with No Output

Most probably you have encountered an error with resource creation, and one of the '**Wait For Resource Creation**' steps is stuck because it cannot find the particular resource it thought existed. In this case, just kill the current script execution and run the steps including and following the failed step. Be careful, while running resource creation step again, you might create more set of resources than you planned.

Moving Large Datasets

During the final steps of deployment when CodeDeploy initiates packaging of Master and Worker application, the deployment process might fail due to its inability to move large sets of data files. This is usually observed when large datasets (approximately in the range of 8-10 GB) are uploaded with the application from your machine. The error message is related to network failure but usually is related to the inability of uploading large amounts of data to S3 from your local machine using the AWS CLI. To work around this issue, we have provided a data-movement.sh script that helps in moving data from your local machine to the worker nodes. The script requires some inputs like the local directory from where to pick up data, the path to AWS Key-Pair file and the Public DNS of EC2 Worker Nodes.

Instance Creation Failures

Using Instance-IDs

Instance-IDs are 17 character IDs of the type i-1234567890abcdef0 that identify an instance uniquely in AWS Cloud. You can use the Instance-ID provided by the deployment script to identify a instance in AWS Web console for debugging any errors.

Existing resources with tags

If there are existing instances with the tags '**Type:aws_mwf_ec2_master**' or '**Type:aws_mwf_ec2_worker**' from the previous run, then the setup would fail. If you are trying to set up a new experiment, then terminating the old instances and starting the deployment script again would resolve this error. If you are trying to push a revision to the same set of instances, then you should make use of the **'push_updated_application'** argument to the deployment script.

Tag creation failure

Although not common, it has been observed that after running the clean infrastructure script and terminating all the instances, if you run the deployment process again, the tagging creation

on instances fails. You can choose to manually tag the instances via the AWS EC2 Web Console.

Failure in Spot instance creation

This usually occurs if your account doesn't have enough Spot Instance limits. Usually, newer accounts might have Spot Instance limits even set to 0 and thus the creation of spot instance might fail for creating even a single instance. If you get an error message saying 'Max Spot Instance Count Exceeded' then you can contact AWS Support for increasing your Spot Instance limits. You can follow our guide for contacting AWS Support described in the Appendix. Also, another common reason for Spot Instance creation failure is the unavailability of Spot Instance, for a given Instance Type in the chosen Availability Zone.

Dry-Run Operation or Not Enough Permission Failure

When we set up a custom IAM user with access restrictions, and try to create a resource that doesn't fit the limited scope of the user, the resource creation step might fail. This failure would usually be caught by the Dry-Run Operation and the error message would point you to the exact error. You can update the policies on the existing user and it would resolve this failure. You can update the policies using the AWS IAM Web console.

Failing running or status checks

The deployment script waits for instances to reach the Running State and pass all Status Checks. Sometimes the instances are launched but fail to either reach the Running State or pass all the Status Checks during the Wait for Event period. In such cases, visit the AWS EC2 Console and check if all the instances are Running and have passed all the Status Checks and then continue executing the next step in the deployment script.

Note

Selecting the right Instance

Instance types are usually selected based on either compute power or storage capacity. The memory capacity of the instance dictates the amount of data you can store in memory at a given point of time without running into **'OutOfMemory'** errors. This consideration can be an excellent parameter for deciding the amount of memory, and subsequently the instance type for your experiment.

Issues with using Spot Instances

Spot Instance is spare computing capacity available with AWS that it provides at a discounted rate. You can reduce your usage cost by opting for Spot Instances instead of On-Demand Instances. Although there are some caveats to know about Spot Instances:
Spot Instances are not always available for all type of instances in each region. You can check the EC2 console to see if a Spot Instance of the given instance type is available for an availability zone.

If the current Spot Instance pricing goes beyond the maximum Spot price set by you when launching the Spot Instance, you can lose access to the Spot Instance. The deployment script uses the On-Demand price as the maximum Spot price for the instance. Thus, the chances of losing your instance are less but not completely impossible.
Although all the accounts on AWS should have access to launch a maximum of 20 Spot Instances of any instance type in any availability zone. As per AWS, the Spot instance limits are dynamic and maybe even smaller than the default 20 Spot Instances. If you get a message saying **'Max Spot Instance Count Exceeded'**, you can contact AWS support for increasing your Spot Instance limits. Please check out the Appendix for more information on contacting AWS. For more information on Spot Instances, follow the link here: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.

Getting the Results

Overview

Once the experiment is completed, run the script **'./get_results.sh'**.
This script will create a result_${date-time} folder in the parent directory with results collected from all the instances, put into individual instance folders. These results are essentially the logs written to by the logger.

Debug Guide

One of the common errors here is, when you run the script, but the script returns **'No such file or directory'** which usually points to the fact that either the application was not running in the first place and thus the logger did not log any data, or the application was started but due to logical errors with the application the logger did not print any results to the file.

Cleaning Infrastructure

Overview

The Infrastructure Cleaning process runs exactly in the opposite order of the deployment process while cleaning all the resources created as a part of the deployment process. Running the script **'./clean_infra.sh all'** cleans all the resources safely. If a certain step causes failure, you can go to the resources' AWS Web Console and delete the resources manually.

Debug Guide

Some of the common errors that one might face during infrastructure cleaning process are:
1. Dry-Run Operation to delete Key Pair/Security Group/IAM Role failed. This error is usually observed when there are instances which are still using the Key Pair/Security Group/IAM Role and you attempt to delete the Key Pair/Security Group/IAM Role.
2. Fatal Error when deleting S3 bucket. This error usually occurs when you try to delete a bucket that has already been deleted.

APPLICATION DEVELOPMENT

Overview

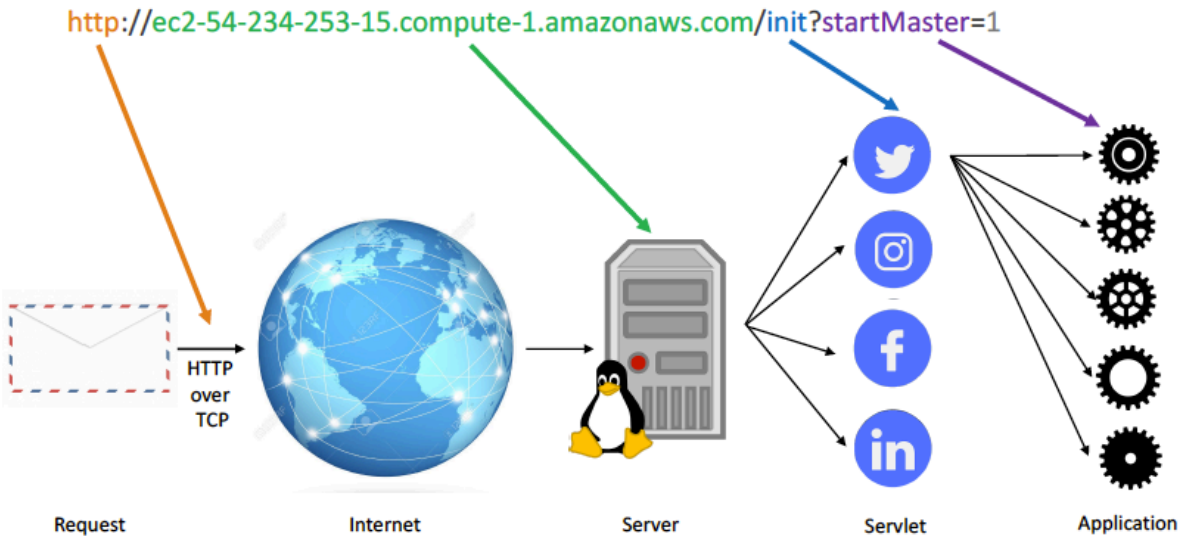

Fig 17: Routing of a typical RESTful API request to a server

The application code is divided into the master_setup and worker_setup directories. Both the Master and Worker are essentially servers running the Spring MVC framework, listening on port 80 for external API calls. The Master node can make RESTful calls to the Worker nodes and the Worker can respond to queries or generate newer RESTful calls back to the Master or other Worker nodes. The examples subdirectory provides some working examples, which will prove to be a great starting point for understanding application development.

Getting Started

Files to Update

There are three important files from the application perspective within both the Worker and Master application.

Server

src/main/java/mc/server/Server.java

Requires least to no modification at all. Sets up the server as a listener on port 80

Servlet

src/main/java/mc/server/servlets/ServletMaster.java

Receives the API calls and routes to appropriate methods defined in AppMaster.java. It is recommended to keep the resource mapping as **'/master'** and **'/worker'** for Master and Worker server respectively. If more than one functionality required, then route the parameters within a resource mapping to different methods using different parameters within a servlet.

Application

src/main/java/mc/server/servlets/AppMaster.java

The methods which service the API calls received by the servlet are written in this file. This file can and should be customized extensively to provide any functionality required by the Master and Worker servers.

Navigating the Non-Application Files

Under every application directory ie. under /master_setup or /worker_setup, the following Non-Application files are present.

1. The docs/ directory provides Javadoc documentation on the Communication and Data Type interfaces designed as a part of the application library.
2. The scripts/ directory contains scripts required for CodeDeploy to install the application dependencies, build the application, execute the application and kill the application safely when a revision needs to be pushed to the nodes.
3. The results/ directory is where the logger writes the results recorded during the application runs. The file type and storage structure are discussed in detail in the Logging section.
4. The data_files/ directory is where the data files specific to the application are stored. They can be read from within the application by providing the complete path to this directory and the file name.
5. The src/main/resources directory contains two critical files.

The first is the configuration file log4j2.xml which is used by Apache Log4j2 logger to define the logger properties. More information about the logger configuration is described in the Logger section.

The other two files are the ec2_master_dns_list and ec2_worker_dns_list. These files contain the Master and the Worker node public DNS' in a comma separated format. These files are populated during the deployment process with the instance details.

Custom Libraries

Custom Libraries have been included with the application to speed up the development of application code.

Datatype Libraries

TypeMatrixInt, TypeMatrixDouble, TypeVectorInt, and TypeVectorDouble are the custom datatype libraries that have been developed to ease the development of applications requiring extensive matrix and vector computations. They provide some standard interfaces, like:

Creating an empty matrix/vector

Creating a randomly initialized matrix/vector

Reading a matrix/vector from a file by providing the row and column separators.

The libraries also provide interface for serialization and deserialization of matrix/vector that greatly eases transferring matrice/vector objects over the network between the different nodes. The details about using these interfaces, including the constructor, method arguments, return types and standard usage has been defined in the Javadoc.

Communication Library

CommAPI library provides a standard interface wrapper around the Unirest library that makes writing GET request, setting the connection timeout, finding the list of active workers in your resource group easier. It also provides a standard interface URL Encoding and Decoding a string before transmitting it over the network. For more information about the Unirest library, please follow the link here: http://unirest.io/java.html.

Logging

Apache Log4j2 framework has been chosen for logging within the application. It provides Asynchronous Rolling File Logging, which can be extremely useful for logging both errors and information.

You can use it to log any experimental results that you want to access later for analysis. They will be made available to the user locally via the get_results.sh script.

The log4j2 configuration file defines some of the standard logger setups. Check the appendix for an explanation about the logger configuration file and how to modify some of the parameters in it. For more information about the Apache Log4j2 utility, follow the link here: https://logging.apache.org/.

Demo Application

In this section, we will dissect the Distributed Matrix-Vector Multiplication Algorithm provided in the examples directory to understand how to implement a distributed algorithm in CREDENCE.

Let us study parts of the Servlet.java file under the master application:

```
Map<String, String[]> paramMap = request.getParameterMap();
for (String param : paramMap.keySet()) {
     if (param.equals("startMaster")) {
          if (paramMap.get(param)[0].equals("1")) {
               AppMaster appMasterObj = new AppMaster();
               appMasterObj.initTask();
          }
     }
}
```

In the above piece of code, we can see that the servlet receives a parameter map with any request. This parameter map key is used to identify which part of the application method should be called to service the request. For a request sent by the user of the type: www.dns.com/master?startMaster=1,

the servlet corresponding to the master tag would receive the parameter map containing the parameter key **'startMaster'** and the parameter value 1. The request could contain any parameter key, and any number of values attached to a key, often separated by an ampersand (&) in the request. Here the received request goes to the method **'initTask()'** of the AppMaster class if the received parameter has a value of 1.

On a side note, the initTask() method plays a crucial role in CREDENCE framework. Once the Master and Worker servers are deployed on the nodes, they need to be triggered to initiate communication, and subsequently, start the experiment. Thus by sending a request that calls the initTask() method, the master starts the experiments. This might involve sending all the workers some requests to initiate computation, or in our case, the vector to be multiplied with matrix splits present at the workers.

A similar servlet structure is set up at the worker end and can be expanded as per the algorithm needs. A general rule for designing a new parameter key is when the Master or the Worker wants to access another node for a service or more specifically a task by calling a certain method on it.

Now looking towards the AppMaster file:

```
final private static Logger logger =
LogManager.getLogger(AppMaster.class);
private List<String> dns_arr_workers =
CommAPI.getEC2DNSList("worker", ",");

public void initTask() {
    TypeVectorInt vect = new TypeVectorInt(100, 0, 100);
     for (String dns : dns_arr_workers) {
        try {
            String vectResp = CommAPI.sendGetRequest("http",
dns, "/worker", new HashMap<String, Object>() {{put("vectIn",
CommAPI.stringURLEncode(vect.serialize(","))); }});
            TypeVectorInt finVect = new TypeVectorInt(100);
            finVect.deserialize(CommAPI.stringURLDecode(vectResp
), ",");
               logger.info(finVect.serialize(","));
          }
          catch (IOException e) {
               logger.error(e.toString());
               e.printStackTrace();
          }
     }
}
```

The first line of the above code setup the logger, which is used to log all the messages and errors. The second line in the above code is used to read the public DNS' of the workers into an ArrayList of String using the file populated during the deployment process.
This initTask() method first created a new random vector containing 100 integer elements with values between 0 and 100. Next, it sends out this vector to all the worker nodes in a for loop one at a time. It waits for the workers to respond with a vector. We serialize the TypeVectorInt object vect to a string and then URLEncode it before sending it over the network to the worker nodes. The received vector needs to go through a reverse process of URLDecoding and is

then deserialized to fill up the TypeVectorInt object finVect. As we can see, we use the logger to log both the received results and the errors in the catch section of the try-catch blog.

It would often be required from a distributed application to be able to send out all the requests simultaneously to all the worker nodes rather than one Worker node at a time. To achieve this, we can launch a separate thread for every new request we send to a Worker node. While we can use, the Thread class interface to create new threads, a better approach provided by Java is to use the ExecutorService. Although a discussion on ExecutorService would be beyond the scope of this guide, one can find more information here: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html.

Next, we will look at the ServletWorker.java file to find how the worker node accepts the vector from the master node.

```
Map<String, String[]> paramMap = request.getParameterMap();
for (String param : paramMap.keySet()) {
     if (param.equals("vectIn")) {
          AppWorker appWorker = new AppWorker();
          prodVect =
appWorker.vectMatMult(paramMap.get(param)[0]);
     }
}
```

Here we see, if the servlet receives a request with a parameter key equal to vectIn, it calls the method vectMatMult on the object of AppWorker class with the received vector as the argument to the vectMatMult method.

Now we can look at the Worker AppWorker.java file to understand how it processes the received vector from the Master node.

```
String vectMatMult(String vectInStr) {
     String vectResp = "";
     try {
          long matProdTimeStart = System.nanoTime();
          TypeMatrixInt mat = new TypeMatrixInt(100, 100, 0,
100);
          TypeVectorInt vectIn = new TypeVectorInt(100);
          vectIn.deserialize(CommAPI.stringURLDecode(vectInStr),
",");
          TypeVectorInt vectFin = new TypeVectorInt(100,
mat.matVectMult(vectIn));
          vectResp =
CommAPI.stringURLEncode(vectFin.serialize(","));
          logger.info("Time Taken to Complete Matrix Vector
Product is " + (System.nanoTime()-matProdTimeStart));
     }
     catch (Exception e){
          logger.error(e.toString());
          e.printStackTrace();
     }
     return vectResp;
}
```

The received vector is deserialized into TypeVectorInt object vectIn which is then multiplied with TypeMatrixInt object mat, to return another TypeVectorInt object vectResp. This vectResp object is the result of the Matrix-Vector Multiplication and is serialized to be returned to the Master node.
Notice how we have used this logger to log the individual Matrix-Vector Multiplication time at the Worker node.

Although this was a simple example of a distributed algorithm, one can extend the application to more complex setups.

Debug Guide

SSH and Restart Master and a Worker

Usually when an application fails to run or returns an error within the logs, then the user can SSH into the master or worker node, kill the running application, restart the application and initiate the experiments again. This would print out the errors to the console if they were not caught in the log. It is highly recommended to print all the errors to the log, as it would ease the identification of errors. Although, because the logger is asynchronous, it might happen that the logger would not log the error before the exception is thrown.

Header Size

In certain cases, when sending extremely long vectors, you could encounter an HTTP Header Exceeded Maximum Allowed Size error. To avoid this issue, it is recommended to modify the Server.java file of Master and Worker application to set the maximum header size using the Undertow Options library. See the following example on how to modify maximum header size of the server:

```
import io.undertow.UndertowOptions;

Undertow server =
Undertow.builder().setServerOption(UndertowOptions.MAX_HEADER_SI
ZE, 10485760)
                  .addHttpListener(80, "0.0.0.0")
                  .setHandler(path)
                  .build();
server.start();
```

The `.setServerOption(UndertowOptions.MAX_HEADER_SIZE, 10485760)` sets the maximum header size, thus resolving the error.

Heap Memory

Sometimes you might encounter the error, java.lang.OutOfMemoryError: Java heap space error. This error is usually encountered when trying to load objects into memory that consume more memory than the allocated heap space for the application. Although you can stretch the heap space allocation by setting the Xmx limits on JVM but is often limited by the memory of the node. Thus, selecting a node with a better memory or running the application with smaller dataset might be the only way to resolve this issue.

Also, make sure that your program is not creating replicated objects which are not necessary for program execution.

Timeout

You might sometimes encounter the **'Connection Timed Out'** error. This error usually occurs when you are executing extremely long running jobs, which is not an uncommon scenario in distributed applications. To overcome these issues, one can use the constructs available in the CommAPI library to set the timeout values within the Master and Worker applications. Also, when sending the curl request to the master node for triggering the experiment, it would be recommended to set the connection timeout to a value large enough for the experiment to complete its execution.

Concurrency

When designing a distributed application, you cannot overlook the importance of synchronization constructs and concurrency errors that can be caused. One must identify the static objects (objects that have only one copy for any object of the class) within the Application and make sure that modifications to these static objects should be synchronized. This is critical when the application is receiving requests from multiple nodes, as these requests might try to

access the static objects within the called method concurrently, and thus synchronizing access to the static object becomes of utmost importance to the accurate execution of the algorithm.

Note

Using Intellij

I recommend using the IntelliJ IDE by JetBrains for your development. It has extremely good static code analysis and code recommendations which greatly improves development. Also, they are generous enough to provide community edition for academic users. Although, any Java based IDE should be able to open the applications and support application development.

APPENDIX

Contacting AWS Support

In case you find yourself in need to contact the AWS Support for increasing the service limits on your account to launch either Spot or On-Demand Instances beyond the default limit of 20 instances, you can follow the guide here to do so: https://docs.aws.amazon.com/awssupport/latest/user/getting-started.html.

Also, if you want to know more about AWS Instance and Service Limits, then check the link here: https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html.

Sample JSON for creating IAM User

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:*",
                "ec2:*",
                "iam:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "codedeploy:*"
            ],
            "Resource": [
                "arn:aws:codedeploy:*:*:application:*",
                "arn:aws:codedeploy:*:*:instance:*",
                "arn:aws:codedeploy:*:*:deploymentgroup:*",
                "arn:aws:codedeploy:*:*:deploymentconfig:*"
            ]
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Deny",
            "Action": "ec2:RunInstances",
            "Resource": [
                "arn:aws:ec2:*:*:image/*",
                "arn:aws:ec2:*:*:subnet/*",
                "arn:aws:ec2:*:*:instance/*",
                "arn:aws:ec2:*:*:volume/*",
                "arn:aws:ec2:*:*:security-group/*",
                "arn:aws:ec2:*:*:network-interface/*"
            ],
            "Condition": {
                "Null": {
                    "aws:RequestTag/Project": "true"
                },
                "StringNotEquals": {
                    "ec2:Region": "us-east-1"
                },
                "NumericGreaterThan": {
                    "ec2:VolumeSize": "20"
                },
                "StringNotLikeIfExists": {
                    "ec2:InstanceType": ["t2.*", "m5.*"]
                }
            }
        }
    ]
}
```

Note

This sample policy creates a user with limited access privileges within your account. The user has access to only, EC2, S3, CodeDeploy and IAM (The first entry in the JSON Policy above shows the services that have been allowed for access to this user) from the wide array of services that AWS provides. It also limits the instance type to only t2.* and m5.* type, the region to us-east-1 and creating an instance with volume size greater than 20GB (The third entry in JSON Policy above shows the deny rules). You can modify the parameters within this sample policy to create another custom user, with more or lesser privileges.

Log4j2 configuration file

log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="debug">
    <Appenders>
        <Console name="Console-Appender" target="SYSTEM_OUT">
            <PatternLayout>
                <pattern>
                    [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} -
%msg%n
                </pattern>>
            </PatternLayout>
        </Console>
        <RollingRandomAccessFile name="Rolling-Random-Access-File-Appender"
                                 fileName="/home/ubuntu/master_setup/results/
results.log"
                                 filePattern="/home/ubuntu/master_setup/resul
ts/results.log.%d{yyyy-MM-dd-hh-mm}.gz">
            <PatternLayout pattern="[%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS}
[%t] %c{1} - %msg%n"/>
            <Policies>
                <SizeBasedTriggeringPolicy size="1 MB"/>
            </Policies>
            <DefaultRolloverStrategy max="30"/>
        </RollingRandomAccessFile>
    </Appenders>
    <Loggers>
        <AsyncRoot level="debug">
            <AppenderRef ref="Console-Appender"/>
            <AppenderRef ref="Rolling-Random-Access-File-Appender"/>
        </AsyncRoot>
    </Loggers>
</Configuration>
```

Note

Here we are using appenders within the logger to log application data. The first appender prints to console, while the second appender prints to a file called results.log in the /results directory of the application. The second appender is a Rolling File Random Access File Appender that provides the capability of archiving the files to a gunzipped format as soon as the log files reach a certain threshold size. This allows to efficiently organize the logs and stops the logs from growing linearly in size. The logs are zipped in the following format: results.log.%d{yyyy-MM-dd-hh-mm}.gz. In case your application is generating logs greater than 1MB in size within a given minute of the experiment's execution, then you can increase the resolution of the log file zip format or the threshold size at which zipping of files occurs or both. Both the appenders are defined as AsyncRoot loggers that allows them to log asynchronously to the file without causing the application to wait for the write to happen to the log file. The logs are written in the following format: [%-5level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} - %msg%n and print any custom message passed to it by the application. Here the %-5level is the level at which the logger was called. For more information about logger levels and how they affect logging please follow the link here: https://logging.apache.org/log4j/2.0/manual/architecture.html.

REFERENCES

Here is a link to the CREDENCE project GitHub repository, containing the source code: https://github.com/MalharSC/aws_master_worker_framework

References Used throughout the Document:

- Unirest Library: http://unirest.io/java.html
- Apache Log4j2 - General: https://logging.apache.org/
- Java Executor Service: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html
- AWS Support: https://docs.aws.amazon.com/awssupport/latest/user/getting-started.html
- AWS Service Limits: https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html
- Apache Log4j2 - Architecture: https://logging.apache.org/log4j/2.0/manual/architecture.html
- AWS Git Secrets: https://github.com/awslabs/git-secrets
- AWS Organizations: https://aws.amazon.com/organizations/
- AWS Docs: Organizations - Creation Guide: https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_create.html
- AWS Docs: Access Credentials: https://docs.aws.amazon.com/general/latest/gr/aws-sec-cred-types.html#access-keys-and-secret-access-keys
- AWS Docs: IAM Role: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html
- AWS Docs: Instance Profile: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2_instance-profiles.html
- AWS Docs: Security Group: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html
- AWS Docs: Key-Pair: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html
- AWS Docs: CodeDeploy - General: https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html
- AWS Docs: CodeDeploy - Instance Configure: https://docs.aws.amazon.com/codedeploy/latest/userguide/instances-ec2-configure.html
- AWS Docs: CodeDeploy - Instance Agent: https://docs.aws.amazon.com/codedeploy/latest/userguide/codedeploy-agent-operations-verify.html
- AWS Docs: Spot Instances: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html

CONTACT

For general information about project CREDENCE and problems with using the CREDENCE platform, please send an email to credence-research@andrew.cmu.edu

For information about contributing to CREDENCE project, please visit our GitHub repository at https://github.com/MalharSC/aws_master_worker_framework and review our Contribution guide.