Jupiter Documentation

ANRG

May 13, 2020

Jupiter Documentation

Jupiter is an Orchestrator for Dispersed Computing that uses Docker containers and Kubernetes (K8s).

The code is open source, and available on GitHub.

The main documentation for the site is organized into a couple sections:

CHAPTER 1

Overview

1.1 Components

Jupiter is an Orchestrator for Dispersed Computing that uses Docker containers and Kubernetes (K8s).

Jupiter enables complex computing applications that are specified as directed acyclic graph (DAG)-based task graphs to be distributed across an arbitrary network of computers in such a way as to optimize the execution of the distributed computations. Depending on the scheduling algorithm/task mapper used with the Jupiter framework, the optimizations may be for different objectives, for example, the goal may be to try and minimize the total end to end delay (makespan) of the computation for a single set of data inputs. Jupiter includes both centralized task mappers such as one that performs the classical HEFT (heterogeneous earliest finish time) scheduling algorithm, as well as an innovative new distributed task mapping framework called WAVE. In order to do enable optimization-oriented task mapping, Jupiter also provides tools for profiling the application run time on the computers as well as profiling and monitoring the performance of the network. Jupiter also provides for container-based code dispatch and execution of the distributed application at run-time for both single-shot and pipelined (streaming) computations.

The Jupiter system has three main components: Profilers, Task Mapper and CIRCE Dispatcher.

- Profilers are tools used to collect information about the system.
  - DRUPE (Network and Resource Profiler) is a tool to collect information about computational resources as well as network links between compute nodes in a dispersed computing system to a central node. DRUPE consists of a network profiler and a resource profiler.
  - The onetime Execution Profiler is a tool to collect information about the computation time of the pipelined computations described in the form of a directed acyclic graph (DAG) on each of the networked computation resources. This tool runs a sample execution of the entire DAG on every node to collect the statistics for each of the task in the DAG as well as the makespan of the entire DAG.
- Task Mapper comes with three different versions: HEFT, WAVE Greedy, WAVE Random; to effciently map the task controllers of a DAG to the processors such that the makespan of the pipelines processing is optimized.
  - HEFT Heterogeneous Earliest Finish Time is a static centralized algorithm for a DAG based task graph that efficiently maps the tasks of the DAG into the processors by taking into account global information about communication delays and execution times.

  - WAVE is a distributed scheduler for DAG type task graph that outputs a mapping of task controllers to real compute nodes by only taking into acount local profiler statistics. Currently we have two types of WAVE algorithms: WAVE Random and WAVE Greedy. WAVE Random is a very simple algorithm that maps the tasks to a random node without taking into acount the profiler data. WAVE Greedy is a Greedy algorithm that uses a weighted sum of different profiler data to map tasks to the most suitable nodes.

- CIRCE is a dispatcher tool for dispersed computing, which can deploy pipelined computations described in the form of a directed acyclic graph (DAG) on multiple geographically dispersed computers (compute nodes). CIRCE uses input and output queues for pipelined execution, and takes care of the data transfer between different tasks. `CIRCE` comes with three different versions: nonpricing scheme, pricing event driven scheme and pricing push scheme.
  - `Nonpricing CIRCE`: static version of dispatcher, which deploys each task controllers on the corresponding compute node given the output of the chosen Task mapper. The task controller is also responsible for the corresponding task itself. This is one-time scheduler. If the user wants to reschedule the compute nodes, he has to run the deploy script again (run corresponding Task mapper and CIRCE again).
  - `Pricing Event driven CIRCE`: dynamic version of dispatcher, which deploys each task controllers on the corresponding compute node given the output of the chosen Task mapper. Moreover, the task controller will select the best current available compute node to perform the task it is responsible for based on the updated resource information (communication delays, execution times, compute resource availability, queue delays at each compute node). The update is performed at the time the task controllers receive the incoming streaming file, the task controllers request the update from the compute nodes.
  - `Pricing Pushing CIRCE`: similar to *Pricing Event driven CIRCE*', but the update is performed in a different way, in which the compute nodes push the update to the task controllers every interval.

The code is open source, and available on GitHub.

1.2 Applications

Jupiter accepts pipelined computations described in a form of a Graph where the main task flow is represented as a Directed Acyclic Graph (DAG). Thus, one should be able separate the graph into two pieces, the DAG part and non-DAG part. Jupiter requires that each task in the DAG part of the graph to be written as a Python function in a separate file under the scripts folder. On the other hand the non-DAG tasks can be either Python function or a shell script with any number of arguments, located under the scripts folder.

As an example, please refer to our codes available for the following applications customized for the Jupiter Orchestrator:

- Coded Network Anomaly Detection : Coded DNAD
- Multi-Camera Processing DAG : MCP DAG
- Automatic-DAG-Generator: Dummy DAG

In order to integrate one specific application into Jupiter, please refer to our documentation regarding Applications.

1.3 Tutorials

There are some provided tutorials:

- Set up kubernetes cluster on Digital Ocean: set up k8s on DO
- Deploy Jupiter on Digital Ocean: Jupiter deployment on DO

- Step by step instructions to set up Jupiter on a private network provided by Sean Griffin (Raytheon BBN Technologies): private network setup

CHAPTER 2

Clone Instructions

This Repository comes with a submodule with links to another repository that contains codes related to one application (Distributed Network Anomaly Detection) of Jupiter.

- If you are interested in cloning just the Jupiter Orchestrator, not the application specific files, run :

```
git clone git@github.com:ANRGUSC/Jupiter.git
```

- If you are interested in cloning the Jupiter Orchestrator along with the Distributed Network Anomaly Detection related files, run :

```
git clone --recurse-submodules git@github.com:ANRGUSC/Jupiter.git
cd Jupiter
git submodule update --remote
```

CHAPTER 3

Requirements

In order to use the Jupiter Orchestrator tool, your computer needs to fulfill the following set of requirements.

- You MUST have `kubectl` installed (instructions here)
- You MUST have `python3` installed
- You MUST have certain python packages (listed in `k8_requirements.txt`) installed. You can install them by simply running

```
pip3 install -r k8_requirements.txt
```

- You MUST have a working kubernetes cluster with `proxy` capability.
- To control the cluster, you need to grab the `admin.conf` file from the k8s master node. When the cluster is bootstrapped by `kubeadm`, the `admin.conf` file is stored in `/etc/kubernetes/admin.conf`. Usually, a copy is made into the `$HOME` folder. Either way, make a copy of `admin.conf` into your local machine's home folder.

Warning: Currently, you need to have `admin.conf` file in the `$HOME` folder. Our python scripts need it exactly there to work.

- Next, you need to run the commands below. You can wrap it up in a script you source or directly place the export line and source line into your `.bashrc` file. However, make sure to re-run the full set of commands if the `admin.conf` file has changed:

```
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf #check if it works with `kubectl get nodes`
source <(kubectl completion bash)
```

- The directory structure of the cloned repo MUST conform with the following:

```
Jupiter
       |   jupiter_config.py
       |   jupiter_config.ini
```

(continues on next page)

(continued from previous page)

```
        |   nodes.txt
        |
        └──profilers
        |
        └──task_mapper
        |
        └──circe
        |
        └──app_specific_files
        |   |
        |   └──APP_folder
        |       |
        |       |   configuration.txt
        |       |   app_config.ini
        |       |
        |       └──scripts
        |       |
        |       └──sample_input
        |
        └──mulhome_scripts
        |___docs
```

CHAPTER 4

Deploy instructions

4.1 Step 1 : Clone Repo

Clone or pull this repo and `cd` into the repo's directory.

4.2 Step 2 : Update Node list

List of nodes for the experiment is kept in file `nodes.txt` (the user needs to fill the file with the appropriate **kubernetes nodenames** of their compute nodes).

You can simply change just the hostnames in the given sample file. The first line should be

```
home nodename
```

Everything else can be the same.

home	nodename
node1	nodename
node2	nodename
node3	nodename

4.3 Step 3 : Setup Home Node

You need to setup the configurations for the circe home, wave home, and the central profiler. For convienice we statically choose a node that will run all dockers. To set that, you have to change the following line in your `jupiter_config.py` file.

```
1 HOME_NODE = 'ubuntu-2gb-ams2-04'
```

This should point to a resource heavy node where you want to run them. We kept it like this for convinience. However in future, this will be made dynamic as well. Next, we also need to point the child of CIRCE master. The CIRCE master is used to dispatch input files to tha DAG. Thus it should point to the ingress tasks of the DAG. Change the following line in the config file to achieve that.

```
HOME_CHILD = 'sample_ingress_task1'
```

For example, in the linked example of Network Monitoring, it is a single task called `localpro`. But if there are multiple ingreass tasks, you have to put all of them by separating them by a `:`.

4.4 Step 4 : Setup APP Folder

You need to make sure that you have a `APP_folder` with all the task specific files inside the `task_specific_files` folder. The `APP_folder` needs to have a `configuration.txt`,‘‘app_config.ini‘‘ and `name_convert.txt`.

The `APP_folder` MUST also contain all executable files of the task graph under the `scripts` sub-folder. You need to follow this exact folder structure to develop an APP for the Jupiter Orchestrator.

Note: Detailed instructions for developing APPs for Jupiter will be posted later.

```
APP_folder
|
|   configuration.txt
|   app_config.ini
|   name_convert.txt
|
└──scripts
|
└──sample_input
```

4.5 Step 5 : Setup the Dockers

4.5.1 Version 2.0 and 3.0

Starting from version 2.0, to simplify the process we have provided with the following scripts:

```
circe/circe_docker_files_generator.py --- prepare Docker files for CIRCE
profilers/execution_profiler/exec_docker_files_generator.py --- for execution profiler
profilers/network_resource_profiler/profiler_docker_files_generator.py --- for DRUPE
task_mapper/heft/heft_docker_files_generator.py --- for HEFT
```

These scripts will read the configuration information from `jupiter_config.ini` and `jupiter_config.py` to help generate corresponding Docker files for all the components.

4.5.2 Version 4.0

The automatic scripts paths are updated:

```
circe/original/circe_docker_files_generator.py --- prepare Docker files for CIRCE␣
↪(nonpricing)
circe/pricing_event/circe_docker_files_generator.py --- prepare Docker files for␣
↪CIRCE (event driven pricing)
circe/pricing_push/circe_docker_files_generator.py --- prepare Docker files for CIRCE␣
↪(pushing pricing)
profilers/execution_profiler_mulhome/exec_docker_files_generator.py --- for execution␣
↪profiler
profilers/network_resource_profiler/profiler_docker_files_generator.py --- for␣
↪network profiler
task_mapper/heft_mulhome/original/heft_docker_files_generator.py --- for HEFT␣
↪(original)
task_mapper/heft_mulhome/modified/heft_docker_files_generator.py --- for HEFT␣
↪(modified)
task_mapper/wave_mulhome/greedy_wave/wave_docker_files_generator.py --- for WAVE␣
↪(greedy)
task_mapper/wave_mulhome/random_wave/wave_docker_files_generator.py --- for WAVE␣
↪(random)
```

4.6 Step 6 : Choose the task mapper

You must choose the Task Mapper from `config.ini`. Currently, there are 4 options from the scheduling algorithm list: centralized (original HEFT, modified HEFT), distributed(random WAVE, greedy WAVE).

```
[CONFIG]
    STATIC_MAPPING = 0
    SCHEDULER = 1

[SCHEDULER_LIST]
    HEFT = 0
    WAVE_RANDOM = 1
    WAVE_GREEDY = 2
    HEFT_MODIFIED = 3
```

Note: When HEFT tries to optimize the Makespan by reducing communication overhead and putting many tasks on the same computing node, it ends up overloading them. While the Jupiter system can recover from failures, multiple failures of the overloaded computing nodes actually ends up adding more delay in the execution of the tasks as well as the communication between tasks due to temporary disruptions of the data flow. The modified HEFT is restricted to allocate no more than `MAX_TASK_ALLOWED` containers per computing node where the number `MAX_TASK_ALLOWED` is dependent upon the processing power of the node. You can find `MAX_TASK_ALLOWED` variable from `heft_dup.py`.

4.7 Step 7 : Optional - Choose the CIRCE dispatcher (only starting from Version 4)

Starting from **Jupiter Version 4**, you must choose the CIRCE dispatcher from `config.ini`. Currently, there are 3 options from the dispatcher list: nonpricing, pricing (event driven scheme, pushing scheme)

4.8 Step 8 : Optional - Modify the File Transfer Method or Network & Resource Monitor Tool

4.8.1 Select File Transfer method

Jupiter by default use `SCP` as the file transfer method. If you want to use any other file transfer tool instead (like `XCP`, etc. . .), you can perform the following 2 steps:

Firstly, refer the *Integration Interface* and write your corresponding File Transfer module.

Secondly, update `config.ini` to make Jupiter use your corresponding File Transfer method.

```
[CONFIG]
TRANSFER = 0

[TRANSFER_LIST]
SCP = 0
```

4.8.2 Select Network & Resource Monitor Tool

Jupiter by default use `DRUPE` as the Network & Resource Monitor Tool. If you want to use any other Network & Resource Monitor Tool, you can perform the following 2 steps:

Firstly, refer the *Integration Interface* and write your corresponding Network & Resource Monitor module.

Secondly, update `config.ini` to make Jupiter use your corresponding Network & Resource Monitor module.

```
[CONFIG]
PROFILER = 0

[PROFILERS_LIST]
DRUPE = 0
```

4.9 Step 9 : Push the Dockers

Now, you need to build your Docker images.

To build Docker images and push them to the Docker Hub repo, first login to Docker Hub using your own credentials by running `docker login`. Starting from **Jupiter Version 2**, we have provided with the following building scripts:

```
scripts/build_push_jupiter.py --- push all Jupiter related dockers
scripts/build_push_circe.py --- Push CIRCE dockers only
scripts/build push_profiler.py --- Push DRUPE dockers only
scripts/build_push_wave.py --- Push WAVE dockers only
scripts/build_push_heft.py --- Push HEFT dockers only
scripts/build_push_exec.py --- Push execution profiler's  dockers only
```

The build path scripts are modified in **Jupiter Version 4**:

```
mulhome_scripts/build_push_jupiter.py --- push all Jupiter related dockers and␣
↪nonpricing circe dispatcher
mulhome_scripts/build_push_pricing_jupiter.py --- push all Jupiter related dockers␣
↪and pricing circe dispatcher
```

(continues on next page)

(continued from previous page)

```
mulhome_scripts/build_push_circe.py --- Push nonpricing CIRCE dockers only
mulhome_scripts/build_push_pricing_circe.py --- Push pricing CIRCE dockers only
mulhome_scripts/build push_profiler.py --- Push DRUPE dockers only
mulhome_scripts/build_push_wave.py --- Push WAVE dockers only
mulhome_scripts/build_push_heft.py --- Push HEFT dockers only
mulhome_scripts/build_push_exec.py --- Push execution profiler's  dockers only
```

Warning: However, before running any of these scripts you should update the `jupiter_config` file with your own docker names as well as dockerhub username. DO NOT run the script without crosschecking the config file.

4.10 Step 10 : Optional - Setup the Proxy (only required for Version 2 & 3)

Now, you have to create a kubernetes proxy. You can do that by running the follwing command on a terminal.

```
kubectl proxy -p 8080
```

4.11 Step 11 : Create the Namespaces

You need to create difference namespaces in your Kubernetes cluster that will be dedicated to the DRUPE, execution profiler, Task Mapper, and CIRCE deployments, respectively. You can create these namespaces commands similar to the following:

```
 kubectl create namespace johndoe-profiler
 kubectl create namespace johndoe-exec
 kubectl create namespace johndoe-mapper
 kubectl create namespace johndoe-circe
```

Warning: You also need to change the respective lines in the `jupiter_config.py` file.

```
DEPLOYMENT_NAMESPACE    = 'johndoe-circe'
PROFILER_NAMESPACE      = 'johndoe-profiler'
MAPPER_NAMESPACE        = 'johndoe-mapper'
EXEC_NAMESPACE          = 'johndoe-exec'
```

4.12 Step 12 : Run the Jupiter Orchestrator

Next, you can simply run:

```
cd mulhome_scripts/
python3 auto_deploy_system.py
```

4.13 Step 13 : Optional - Alternate scheduler

If you do not want to use our task mappers (`HEFT` or `WAVE`) for the scheduler and design your own, you can do that by simply using the `static_assignment.py`. You must do that by setting `STATIC_MAPPING` to `1` from `jupiter_config.ini`. You have to pipe your scheduling output to the `static_assignment.py` while conforming to the sample dag and sample schedule structure. Then you can run:

```
cd mulhome_scripts/
python3 auto_deploy_system.py
```

4.14 Step 14 : Interact With the DAG

Now you can interact with the pos using the kubernetes dashboard. To access it just pen up a browser on your local machine and go to `http://127.0.0.1:8080/ui`. You should see the k8s dashboard. Hit `Ctrl+c` on the terminal running the server to turn off the proxy.

CHAPTER 5

Teardown

To teardown the DAG deployment, run the following:

```
python3 k8s_jupiter_teardown.py
```

Once the deployment is torn down, you can simply start from the begining of these instructions to make changes to your code and redeploy the DAG.

CHAPTER 6

Integration Interface

Jupiter by default use `SCP` as the file transfer method and `DRUPE` as the network monitoring tool. We have decoupled these modules in Jupiter so that you can create and use your own modules if you want.

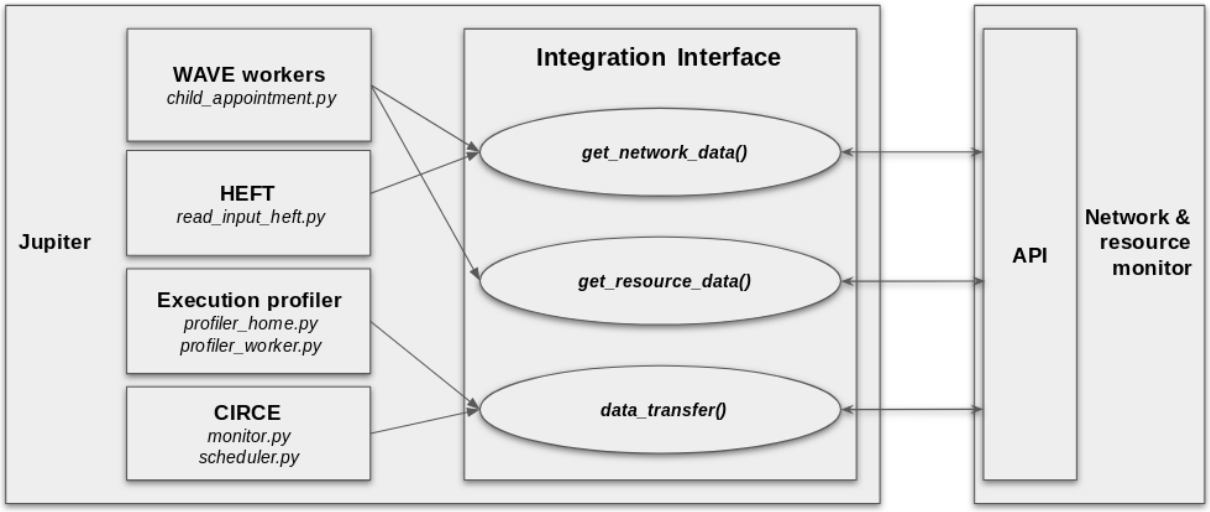


Fig. 1: General Integration Interface between Jupiter and the Network & Resource Monitor Tool

The Jupiter integration interface use the following methods:

- `get_network_data()` : retrieve network data from the Network & Resource Monitor Tool.
- `get_resource_data()` : retrieve resource data from the Network & Resource Monitor Tool.
- `data_transfer(IP,user,pword,resource,destination)`: transfer file to the destination node with provided information (IP,username,password) and file paths (source, destination).

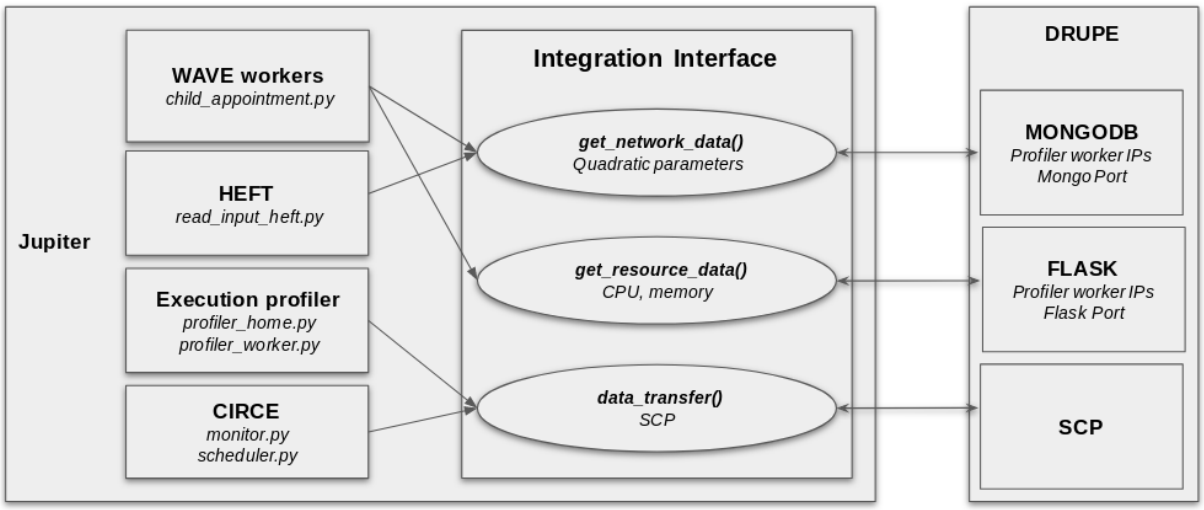


Fig. 2: Example of SCP and DRUPE using the Integration Interface

Warning: The network data from the Network & Resource Monitor Tool working with Jupiter must be in the format of `quadratic parameters` which can specify the communication cost of the network links.

Warning: The resource data from the Network & Resource Monitor Tool working with Jupiter must be the combination of `CPU and memory`.

Please follow the following guideline (with the examples from `SCP` and `DRUPE`) to map your specific modules to the corresponding methods of the interface.

6.1 File Transfer method

Write the data transfer function. In `SCP` example, the function is `data_transfer_scp`. Add the corresponding mapping part for data transfer function:

```
def transfer_mapping_decorator(TRANSFER):
        def data_transfer_scp(IP,user,pword,source, destination):
            retry = 0
            while retry < num_retries:
                try:
                    cmd = "sshpass -p %s scp -P %s -o StrictHostKeyChecking=no -r %s
↪%s@%s:%s" % (pword, ssh_port, source, user, IP, destination)
                    os.system(cmd)
                    print('data transfer complete\n')
                    break
                except:
                    print('profiler_worker.txt: SSH Connection refused or File␣
↪transfer failed, will retry in 2 seconds')
                    time.sleep(2)
                    retry += 1
```

(continues on next page)

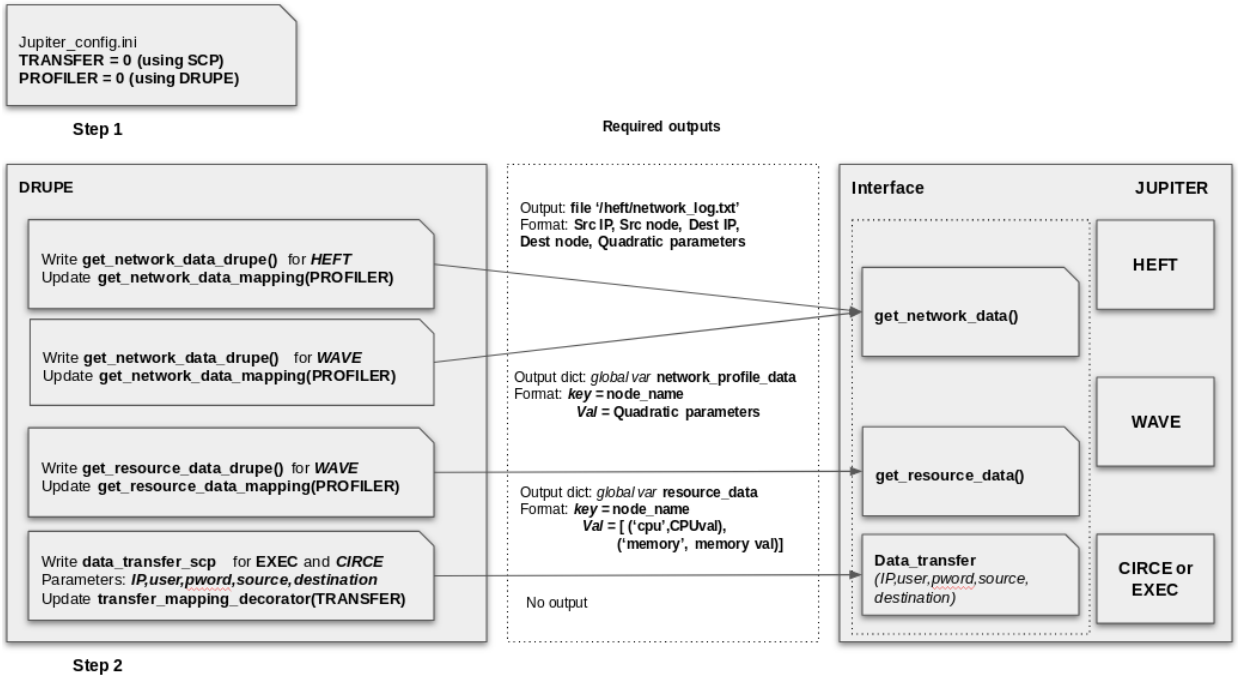


Fig. 3: The example with more detailed implementation.

(continued from previous page)

```
14
15          if TRANSFER==0:
16              return data_transfer_scp
17          return data_transfer_scp
```

6.2 Network & Resource Monitor Tool

6.2.1 Get resource data

Write the resource data crawling function. In `DRUPE` example, the function is `get_resource_data_drupe`. Add the corresponding mapping part for resource data crawling function:

```
1 def get_resource_data_mapping(PROFILER=0):
2         if PROFILER==0:
3             return profilers_mapping_decorator(get_resource_data_drupe)
4         return profilers_mapping_decorator(get_resource_data_drupe)
```

6.2.2 Get network data

Write the network data crawling function. In `DRUPE` example, the function is `get_network_data_drupe`. Add the corresponding mapping part for network data crawling function:

```
1 def get_network_data_mapping(PROFILER=0):
2         if PROFILER==0:
```

(continues on next page)

(continued from previous page)

```
            return profilers_mapping_decorator(get_network_data_drupe)
        return profilers_mapping_decorator(get_network_data_drupe)
```

CHAPTER 7

Jupiter Evaluation

We have built a runtime profiler for the purpose of gathering and analyzing relevant performance statistics while running Jupiter. The runtime profiler's purpose is to collect timestamp information either for all the tasks, compute nodes and incoming files, or file transfer process. This built-in runtime profiler runs at the same time with the deployed system without any further actions.

7.1 Automatic evaluation script

We wrote the optional script `evaluate.py` inside CIRCE , which you can choose to run or not in `start_home.sh`. This script is only for testing purpose of Coded DNAD (Coded Network Anomaly Detection Application). After choosing this option and build the corresponding image, the script will be started automatically after `CIRCE` finishes dispatching all the tasks on all the computing nodes.

7.2 Collecting task performance statistics

This part of runtime profiler is integrated with CIRCE , the dispatcher tool of Jupiter You can find corresponding runtime profiler implementation in `monitor.py` and `scheduler.py` scripts of `CIRCE`. Every compute node has a Flask server which can send related runtime information to the Flask server on the home node of `CIRCE`. All the task-related runtime statistics are output at `runtime_tasks.txt` on the `CIRCE home node` under the following collumn-based format: `Task_name`, `local_input_file`, `Enter_time`, `Execute_time`, `Finish_time`, `Elapse_time`, `Duration_time`, `Waiting_time`. This runtime file is only available after the `evaluation.py` finishes running.

- Enter time: time the input file enter the queue
- Execute time: time the input file is processed
- Finish time: time the output file is generated
- Elapse time: total time since the input file is created till the output file is created
- Duration time: total execution time of the task

- Waiting time: total time since the input file is created till it is processed

7.3 Collecting file transfer performance statistics

This part of runtime profiler is implemented with the integration interface Jupiter, which servers the purpose of collecting runtime information of different data transfer methods in order for comparison. You can find corresponding runtime profiler implementation in ‘‘CIRCE‘‘integration inferface.

- There are two options, whether collecting statistics only in the senders, or in both the senders and the receivers. This can be configured in `jupiter_config.ini` as the `RUNTIME` variable.
- All the data transfer related runtime statistics of the senders are output at `runtime_transfer_sender.txt` on each computing node under the following collumn-based format: `Node_name, Transfer_Type, File_path, Time_stamp`.
- All the data transfer related runtime statistics of the receivers are output at `runtime_transfer_receivers.txt` on each computing node under the following collumn-based format: `Node_name, Transfer_Type, File_path, Time_stamp`.
- If the value of `Time_stamp` is `-1`, it indicates that the file has not been transfered successfully; else it indicates the timestamp that the file starts to be transfered and the file has been transfered successfully.

CHAPTER 8

Project Structure

The directory structure of the project MUST conform with the following:

```
Jupiter/
├── jupiter_config.py
├── jupiter_config.ini
├── k8_requirements.txt
├── LICENSE.txt
├── nodes.txt
├── ...
├── docs
├── circe
│   ├── original
|   |   ├── home_node.Dockerfile
|   |   ├── circe_docker_files_generator.py
|   |   ├── monitor.py
|   |   ├── readconfig.py
|   |   ├── requirements.txt
|   |   ├── evaluate.py
|   |   ├── runtime_profiler_mongodb
|   |   ├── scheduler.py
|   |   ├── start_home.sh
|   |   ├── start_worker.sh
|   |   └── worker_node.Dockerfile
│   ├── pricing_event
|   |   ├── home_node.Dockerfile
|   |   ├── circe_docker_files_generator.py
|   |   ├── monitor.py
|   |   ├── readconfig.py
|   |   ├── requirements.txt
|   |   ├── evaluate.py
|   |   ├── runtime_profiler_mongodb
|   |   ├── scheduler.py
|   |   ├── compute.py
|   |   ├── start_home.sh
|   |   ├
```

(continues on next page)

(continued from previous page)

```
|   |   |   ├── start_computing_worker.sh
|   |   |   ├── start_controller_worker.sh
|   |   |   ├── controller_worker_node.Dockerfile
|   |   |   └── computing_worker_node.Dockerfile
|   |   ├── pricing_push
|   ├── task_mapper
|   |
|   heft_mulhome
|   |__original
|   |   |   ├── heft.Dockerfile
|   |   |   ├── heft_dockerfile_generator.py
|   |   |   ├── heft_dup.py
|   |   |   ├── master_heft.py
|   |   |   ├── read_input_heft.py
|   |   |   ├── requirements.txt
|   |   |   ├── start.sh
|   |   |   ├── write_input_heft.py
|   |__modified
|   |   |   ├── heft.Dockerfile
|   |   |   ├── heft_dockerfile_generator.py
|   |   |   ├── heft_dup.py
|   |   |   ├── master_heft.py
|   |   |   ├── read_input_heft.py
|   |   |   ├── requirements.txt
|   |   |   ├── start.sh
|   |   |   ├── write_input_heft.py
└── wave_mulhome
|   ├── random_wave
|   |   |   ├── home
|   |   |   |   ├── requirements.txt
|   |   |   |   ├── master_random.py
|   |   |   |   └── start.sh
|   |   |   └── worker
|   |   |   |   ├── child_appointment_random.py
|   |   |   |   ├── requirements.txt
|   |   |   |   └── start.sh
|   |   |   |__ home.Dockerfile
|   |   |   |__ worker.Dockerfile
|   |   |   |__ wave_docker_files_generator.py
|   ├── greedy_wave
|   |   |   ├── home
|   |   |   |   ├── requirements.txt
|   |   |   |   ├── master_greedy.py
|   |   |   |   └── start.sh
|   |   |   └── worker
|   |   |   |   ├── child_appointment_greedy.py
|   |   |   |   ├── requirements.txt
|   |   |   |   └── start.sh
|   |   |   |__ home.Dockerfile
|   |   |   |__ worker.Dockerfile
|   |   |   |__ wave_docker_files_generator.py
├── profilers
|   ├── network_resource_profiler_mulhome
|   |   |___home
|   |   |   |   |
|   |   |   |   ├── central_input
|   |   |   |   |   ├── link_list.txt
```

(continues on next page)

(continued from previous page)

```
|   |   |   |   └── nodes.txt
|   |   |   ├── central_mongod
|   |   |   ├── central_query_statistics.py
|   |   |   ├── central_scheduler.py
|   |   |   ├── generate_link_list.py
|   |   |   ├── requirements.txt
|   |   |   ├── resource_profiling_files
|   |   |   |   ├── insert_to_container.py
|   |   |   |   ├── ip_path
|   |   |   |   ├── job.py
|   |   |   |   └── read_info.py
|   |   |   └── start.sh
|   |   |___|___worker
|   |   |   |   |
|   |   |   |   ├── automate_droplet.py
|   |   |   |   ├── droplet_generate_random_files
|   |   |   |   ├── droplet_mongod
|   |   |   |   ├── droplet_scp_time_transfer
|   |   |   |   ├── requirements.txt
|   |   |   |   ├── resource_profiler.py
|   |   |   |   └── start.sh
|   |   |   |___profiler_docker_files_generator.py
|   |   |   |___profiler_home.Dockerfile
|   |   |   |___profiler_worker.Dockerfile
|   |   ├── execution_profiler_mulhome
|   |   |   |___exec_docker_files_generator.py
|   |   |   |___exec_home.Dockerfile
|   |   |   |___exec_worker.Dockerfile
|   |   |   |___get_files.py
|   |   |   |___profiler_home.py
|   |   |   |___profiler_worker.py
|   |   |   |___requirements.txt
|   |   |   |___start_home.sh
|   |   |   |___start_worker.sh
├── task_specific_files
|   |   └── APP_Folder
|   |       ├── configuration.txt
|   |       ├── app_config.ini
|   |       ├── name_convert.txt
|   |       ├── sample_input
|   |       |   ├── sample1
|   |       |   └── sample2
|   |       └── scripts
|   |           ├── task1.py
|   |           └── task2.py
|
└── mulhome_scripts
    ├── auto_deploy_system.py
    ├── auto_teardown_system.py
    ├── build_push_circe.py
    ├── build_push_pricing_circe.py
    ├── build_push_jupiter.py
    ├── build_push_profiler.py
    ├── build_push_wave.py
    ├── build_push_heft.py
    ├── build_push_exec.py
    ├── delete_all_circe.py
```

(continues on next page)

(continued from previous page)

```
    ├── delete_all_pricing_circe.py
    ├── delete_all_profilers.py
    ├── delete_all_waves.py
    ├── delete_all_heft.py
    ├── delete_all_exec.py
    ├── k8s_circe_scheduler.py
    ├── k8s_pricing_circe_scheduler.py
    ├── k8s_heft_scheduler.py
    ├── k8s_exec_scheduler.py
    ├── k8s_profiler_scheduler.py
    ├── k8s_wave_scheduler.py
    ├── static_assignment.py
    ├── utilities.py
    ├── keep_alive.py
    ├── write_circe_service_specs.py
    ├── write_circe_specs.py
    ├── write_pricing_circe_service_specs.py
    ├── write_pricing_circe_specs.py
    ├── write_profiler_service_specs.py
    ├── write_profiler_specs.py
    ├── write_wave_service_specs.py
    ├── write_wave_specs.py
    ├── write_heft_service_specs.py
    ├── write_heft_specs.py
    ├── write_wave_service_specs.py
    └── write_wave_specs.py
```

CHAPTER 9

References

[1] Quynh Nguyen, Pradipta Ghosh, and Bhaskar Krishnamachari, **End to End Network Performance Monitoring for Dispersed Computing** , International Conference on Computing, Networking and Communications, March 2018

[2] Aleksandra Knezevic, Quynh Nguyen, Jason A. Tran, Pradipta Ghosh, Pranav Sakulkar, Bhaskar Krishnamachari, and Murali Annavaram, **DEMO: CIRCE – A runtime scheduler for DAG-based dispersed computing,** The Second ACM/IEEE Symposium on Edge Computing (SEC) 2017. (poster)

[3] Pranav Sakulkar, Pradipta Ghosh, Aleksandra Knezevic, Jiatong Wang, Quynh Nguyen, Jason Tran, H.V. Krishna Giri Narra, Zhifeng Lin, Songze Li, Ming Yu, Bhaskar Krishnamachari, Salman Avestimehr, and Murali Annavaram, **WAVE: A Distributed Scheduling Framework for Dispersed Computing** , USC ANRG Technical Report, ANRG-2018-01.

CHAPTER 10

Applications

10.1 Quick Integration

In order to integrate your applications with Jupiter, there are some requirements in format and syntax for the inputs, outputs and task scripts.

Files `nodes.txt`, `jupiter_config.py` and `config.ini` are found in the main folder of Jupiter and should be updated based on the requirements of users. Files `app_config.ini`, `configurations.txt` and `name_convert.txt` should be present in the application folder to support different Jupiter modules to work properly.

10.1.1 Input - File nodes.txt

This file lists all the nodes, line by line, in the following format:

home	nodename
node2	nodename
node3	nodename
node4	nodename

A given sample of node file:

Warning: There are 2 kind of nodes: `home node` and `compute node`. Home nodes start with `home`, which allow stream of incoming data for the given DAG tasks as well as store all statistical performance information. Compute nodes start with `node`, which will perform scheduling tasks and compute tasks. In the case of nonpricing CIRCE dispatcher, only 1 home node is supported. In the case of pricing CIRCE dispatcher (Event driven or Pushing scheme), multiple home nodes are supported. The home node list should be as followed: `home`, `home2`, `home3`.

```
home ubuntu-2gb-nyc2-01-fra1
node2 ubuntu-2gb-sgp1-01-blr1
node3 ubuntu-1gb-sfo1-01-nyc1
node4 ubuntu-1gb-fra1-01-fra1
node5 ubuntu-1gb-lon1-01-sgp1
node6 ubuntu-1gb-ams2-01-ams3
node7 ubuntu-1gb-sfo2-01-sfo2
node8 ubuntu-1gb-nyc1-01-lon1
node9 ubuntu-2gb-lon1-01-nyc3
node10 ubuntu-2gb-fra1-01-nyc1
node11 ubuntu-2gb-fra1-02-fra1
node12 ubuntu-2gb-tor1-01-nyc1
node13 ubuntu-2gb-tor1-02-ams3
node14 ubuntu-2gb-tor1-03-lon1
node15 ubuntu-2gb-sfo2-01-blr1
node16 ubuntu-2gb-sfo2-02-tor1
node17 ubuntu-2gb-sfo2-03-blr1
node18 ubuntu-2gb-sfo2-05-fra1
node19 ubuntu-2gb-blr1-01-fra1
node20 ubuntu-2gb-blr1-02-sgp1
node21 ubuntu-2gb-blr1-03-ams3
node22 ubuntu-2gb-blr1-04-nyc1
node23 ubuntu-2gb-sgp1-01-nyc3
node24 ubuntu-2gb-fra1-01-nyc1
node25 ubuntu-2gb-fra1-02-sgp1
```

10.1.2 Input - File jupiter_config.py

This file includes all paths configuration for Jupiter system to start. The latest version of jupiter configuration file:

```
global STATIC_MAPPING, SCHEDULER, TRANSFER, PROFILER, RUNTIME, PRICING, PRICE_OPTION

STATIC_MAPPING          = int(config['CONFIG']['STATIC_MAPPING'])
# scheduler option chosen from SCHEDULER_LIST
SCHEDULER               = int(config['CONFIG']['SCHEDULER'])
# transfer option chosen from TRANSFER_LIST
TRANSFER                = int(config['CONFIG']['TRANSFER'])
# Network and Resource profiler (TA2) option chosen from TA2_LIST
PROFILER                = int(config['CONFIG']['PROFILER'])
# Runtime profiling for data transfer methods: 0 for only senders, 1 for both senders␣
↪and receivers
RUNTIME                 = int(config['CONFIG']['RUNTIME'])
# Using pricing or original scheme
PRICING                 = int(config['CONFIG']['PRICING'])
# Pricing option from pricing option list
PRICE_OPTION          = int(config['CONFIG']['PRICE_OPTION'])

"""Authorization information in the containers"""
global USERNAME, PASSWORD

USERNAME                = config['AUTH']['USERNAME']
PASSWORD                = config['AUTH']['PASSWORD']

"""Port and target port in containers for services to be used: Mongo, SSH and Flask"""
global MONGO_SVC, MONGO_DOCKER, SSH_SVC, SSH_DOCKER, FLASK_SVC, FLASK_DOCKER

MONGO_SVC               = config['PORT']['MONGO_SVC']
MONGO_DOCKER            = config['PORT']['MONGO_DOCKER']
SSH_SVC                 = config['PORT']['SSH_SVC']
SSH_DOCKER              = config['PORT']['SSH_DOCKER']
FLASK_SVC               = config['PORT']['FLASK_SVC']
FLASK_DOCKER            = config['PORT']['FLASK_DOCKER']

"""Modules path of Jupiter"""
global NETR_PROFILER_PATH, EXEC_PROFILER_PATH, CIRCE_PATH, HEFT_PATH, WAVE_PATH,␣
↪SCRIPT_PATH, CIRCE_ORIGINAL_PATH

# default network and resource profiler: DRUPE
# default wave mapper: random wave
NETR_PROFILER_PATH      = HERE + 'profilers/network_resource_profiler/'
EXEC_PROFILER_PATH      = HERE + 'profilers/execution_profiler/'
CIRCE_PATH              = HERE + 'circe/pricing/'
HEFT_PATH               = HERE + 'task_mapper/heft/original/'
WAVE_PATH               = HERE + 'task_mapper/wave/random_wave/'
SCRIPT_PATH             = HERE + 'scripts/'

global mapper_option
mapper_option           = 'heft'


if SCHEDULER == int(config['SCHEDULER_LIST']['WAVE_RANDOM']):
    print('Task mapper: Wave random selected')
    WAVE_PATH           = HERE + 'task_mapper/wave/random_wave/'
    mapper_option       = 'random'
```

(continues on next page)

(continued from previous page)

```
elif SCHEDULER == int(config['SCHEDULER_LIST']['WAVE_GREEDY']):
    print('Task mapper: Wave greedy selected')
    WAVE_PATH           = HERE + 'task_mapper/wave/greedy_wave/'
    mapper_option       = 'greedy'
elif SCHEDULER == int(config['SCHEDULER_LIST']['HEFT_MODIFIED']):
    print('Task mapper: Heft modified selected')
    HEFT_PATH           = HERE + 'task_mapper/heft/modified/'
    mapper_option       = 'modified'
else:
    print('Task mapper: Heft original selected')

global pricing_option, profiler_option

pricing_option          = 'pricing' #original pricing
profiler_option         = 'onehome'

if PRICING == 1:#multiple home (push circe)
    pricing_option      = 'pricing_push'
    profiler_option     = 'multiple_home'
    NETR_PROFILER_PATH  = HERE + 'profilers/network_resource_profiler_mulhome/'
    EXEC_PROFILER_PATH  = HERE + 'profilers/execution_profiler_mulhome/'
    HEFT_PATH           = HERE + 'task_mapper/heft_mulhome/original/'
    WAVE_PATH           = HERE + 'task_mapper/wave_mulhome/greedy_wave/'
    print('Pricing pushing scheme selected')
if PRICING == 2:#multiple home, pricing (event-driven circe)
    pricing_option      = 'pricing_event'
    profiler_option     = 'multiple_home'
    NETR_PROFILER_PATH  = HERE + 'profilers/network_resource_profiler_mulhome/'
    EXEC_PROFILER_PATH  = HERE + 'profilers/execution_profiler_mulhome/'
    HEFT_PATH           = HERE + 'task_mapper/heft_mulhome/original/'
    WAVE_PATH           = HERE + 'task_mapper/wave_mulhome/greedy_wave/'
    print('Pricing event driven scheme selected')

CIRCE_PATH              = HERE + 'circe/%s/'%(pricing_option)
if PRICING == 0: #non-pricing
    CIRCE_PATH          = HERE + 'circe/original/'
    NETR_PROFILER_PATH  = HERE + 'profilers/network_resource_profiler_mulhome/'
    EXEC_PROFILER_PATH  = HERE + 'profilers/execution_profiler_mulhome/'
    HEFT_PATH           = HERE + 'task_mapper/heft_mulhome/original/'
    WAVE_PATH           = HERE + 'task_mapper/wave_mulhome/greedy_wave/'
    print('Non pricing scheme selected')
"""Kubernetes required information"""
global KUBECONFIG_PATH, DEPLOYMENT_NAMESPACE, PROFILER_NAMESPACE, MAPPER_NAMESPACE,␣
↪EXEC_NAMESPACE

KUBECONFIG_PATH         = os.environ['KUBECONFIG']

# Namespaces
DEPLOYMENT_NAMESPACE    = 'johndoe-circe'
PROFILER_NAMESPACE      = 'johndoe-profiler'
MAPPER_NAMESPACE        = 'johndoe-mapper'
EXEC_NAMESPACE          = 'johndoe-exec'

""" Node file path and first task information """
global HOME_NODE, HOME_CHILD

HOME_NODE               = get_home_node(HERE + 'nodes.txt')
```

(continues on next page)

(continued from previous page)

```
HOME_CHILD              = 'sample_ingress_task1'

"""pricing CIRCE home and worker images"""
global PRICING_HOME_IMAGE, WORKER_CONTROLLER_IMAGE, WORKER_COMPUTING_IMAGE

PRICING_HOME_IMAGE      = 'docker.io/johndoe/%s_circe_home:coded' %(pricing_option)
WORKER_CONTROLLER_IMAGE = 'docker.io/johndoe/%s_circe_controller:coded' %(pricing_
↪option)
WORKER_COMPUTING_IMAGE  = 'docker.io/johndoe/%s_circe_computing:coded' %(pricing_
↪option)

"""CIRCE home and worker images for execution profiler"""
global HOME_IMAGE, WORKER_IMAGE

HOME_IMAGE              = 'docker.io/johndoe/circe_home:coded'
WORKER_IMAGE            = 'docker.io/johndoe/circe_worker:coded'

"""DRUPE home and worker images"""
global PROFILER_HOME_IMAGE, PROFILER_WORKER_IMAGE

PROFILER_HOME_IMAGE     = 'docker.io/johndoe/%s_profiler_home:coded'%(profiler_option)
PROFILER_WORKER_IMAGE   = 'docker.io/johndoe/%s_profiler_worker:coded'%(profiler_
↪option)

"""WAVE home and worker images"""
global WAVE_HOME_IMAGE, WAVE_WORKER_IMAGE

#coded: random, v1: greedy

WAVE_HOME_IMAGE         = 'docker.io/johndoe/%s_%s_wave_home:coded' %(mapper_option,
↪profiler_option)
WAVE_WORKER_IMAGE       = 'docker.io/johndoe/%s_%s_wave_worker:coded' %(mapper_option,
↪profiler_option)

"""Execution profiler home and worker images"""
global EXEC_HOME_IMAGE, EXEC_WORKER_IMAGE


EXEC_HOME_IMAGE         = 'docker.io/johndoe/%s_exec_home:coded'%(profiler_option)
EXEC_WORKER_IMAGE       = 'docker.io/johndoe/%s_exec_worker:coded'%(profiler_option)

"""HEFT docker image"""
global HEFT_IMAGE

HEFT_IMAGE              = 'docker.io/johndoe/%s_heft:coded'%(profiler_option)

"""Application Information"""
global APP_PATH, APP_NAME

APP_PATH                = HERE  + 'app_specific_files/network_monitoring_app/'
APP_NAME                = 'app_specific_files/network_monitoring_app'
```

Warning: You need to create required namespaces in your Kubernetes cluster that will be dedicated to the profiler, scheduling mapper (to choose specific scheduling algorithms from HEFT, Random WAVE, greedy WAVE), and CIRCE deployments (non-pricing, pricing event driven or pricing push), respectively. You also need to update

your namespace information correspondingly.

```
    DEPLOYMENT_NAMESPACE    = 'johndoe-circe'
    PROFILER_NAMESPACE      = 'johndoe-profiler'
    MAPPER_NAMESPACE        = 'johndoe-mapper'
    EXEC_NAMESPACE          = 'johndoe-exec'
```

You also need to specify the corresponding information:

- CIRCE images : `HOME_IMAGE` and `WORKER_IMAGE`
- Pricing CIRCE images : `PRICING_HOME_IMAGE`, `WORKER_CONTROLLER_IMAGE` and `WORKER_COMPUTING_IMAGE`
- DRUPE images : `PROFILER_HOME_IMAGE` and `PROFILER_WORKER_IMAGE`
- Execution profiler images: `EXEC_HOME_IMAGE` and `EXEC_WORKER_IMAGE`
- HEFT images: `HEFT_IMAGE`
- WAVE images : `WAVE_HOME_IMAGE` and `WAVE_WORKER_IMAGE`
- Initial task : `HOME_CHILD`
- The application folder : `APP_PATH`. The tasks specific files is recommended to be put in the folder `task_specific_files`.

10.1.3 Input - File config.ini

This file includes all configuration options for Jupiter system to start. The latest version of `config.ini` file includes types of mapping (static or dynamic), port information (SSH, Flask, Mongo), authorization (username and password), scheduling algorithm (HEFT original, random WAVE, greedy WAVE, HEFT modified):

```
[CONFIG]
    STATIC_MAPPING = 0
    SCHEDULER = 1
    TRANSFER = 0
    PROFILER = 0
    RUNTIME = 1
    PRICING = 1
    PRICE_OPTION = 0
[PORT]
    MONGO_SVC = 6200
    MONGO_DOCKER = 27017
    SSH_SVC = 5000
    SSH_DOCKER = 22
    FLASK_SVC = 6100
    FLASK_DOCKER = 8888
[AUTH]
    USERNAME = root
    PASSWORD = PASSWORD
[OTHER]
    MAX_LOG = 10
    SSH_RETRY_NUM = 20
    TASK_QUEUE_SIZE = -1
[SCHEDULER_LIST]
    HEFT = 0
```

(continues on next page)

(continued from previous page)

```
    WAVE_RANDOM = 1
    WAVE_GREEDY = 2
    HEFT_MODIFIED = 3
[PROFILERS_LIST]
    DRUPE = 0
[TRANSFER_LIST]
    SCP = 0
[PRICING_LIST]
    NONPRICING = 0
    PUSH_MULTIPLEHOME = 1
    DRIVEN_MULTIPLEHOME = 2
[PRICING_FUNCTION_LIST]
    SUM = 1
    MAX = 2
```

Warning: You can specify the following values:

- `PRICING` in `CONFIG` section to choose the specific CIRCE dispatcher from the `PRICING_LIST`. There are three kinds of CIRCE dispatcher: `NONPRICING`, `PUSH_MULTIPLEHOME` and `DRIVEN_MULTIPLEHOME`.
- `SCHEDULER` in `CONFIG` section to choose the specific scheduling algorithm from the `SCHEDULER_LIST`. `STATIC_MAPPING` is only chosen on testing purpose.
- `PROFILER` in `CONFIG` section to choose the specific network monitoring from the `PROFILERS_LIST`. The default network monitoring tool that we used is `DRUPE`. If you want to use another network monitoring tool, please refer to the guideline how to use the interface.
- `TRANSFER` in `CONFIG` section to choose the specific file transfer method for Jupiter from the `TRANSFER_LIST`. The default file transfer method that we used is `SCP`. If you want to use another file transfer method, please refer to the guideline how to use the interface.

10.1.4 Input - File configuration.txt

The tasks specific files is recommended to be put in the folder `task_specific_files`. Inside the application folder, there should be a `configuration.txt` file having the DAG description. First line is an integer which gives the number of lines the DAG is taking. DAG is represented in the form of adjacency list:

```
parent_task NUM_INPUTS FLAG child_task1 child_task2 child task3 ...
```

- `parent_task` is the name of the parent task
- `NUM_INPUTS` is an integer representing the number of input files the task needs in order to start processing (some tasks could require more than input).
- `FLAG` is `true` or `false`. Based on its value, `monitor.py` will either send a single output of the task to all its children (when true), or it will wait the output files and start putting them into queue (when false). Once the queue size is equal to the number of children, it will send one output to one child (first output to first listed child, etc.).
- `child_task1`, `child_task2`, `child_task3...` are the names of child tasks of the current parent task.

A given sample of application configuration file:

```
41
localpro 1 false aggregate0 aggregate1 aggregate2
aggregate0 1 true simpledetector0 astutedetector0 dftdetector0 teradetector0
aggregate1 1 true simpledetector1 astutedetector1 dftdetector1 teradetector1
aggregate2 1 true simpledetector2 astutedetector2 dftdetector2 teradetector2
simpledetector0 1 true fusioncenter0
simpledetector1 1 true fusioncenter1
simpledetector2 1 true fusioncenter2
astutedetector0 1 true fusioncenter0
astutedetector1 1 true fusioncenter1
astutedetector2 1 true fusioncenter2
dftdetector0 1 true fusioncenter0 dftslave00 dftslave01 dftslave02
dftdetector1 1 true fusioncenter1 dftslave10 dftslave11 dftslave12
dftdetector2 1 true fusioncenter2 dftslave20 dftslave21 dftslave22
dftslave00 1 false dftslave00
dftslave01 1 false dftslave01
dftslave02 1 false dftslave02
dftslave10 1 false dftslave10
dftslave11 1 false dftslave11
dftslave12 1 false dftslave12
dftslave20 1 false dftslave20
dftslave21 1 false dftslave21
dftslave22 1 false dftslave22
teradetector0 1 true fusioncenter0 teramaster0
teradetector1 1 true fusioncenter1 teramaster1
teradetector2 1 true fusioncenter2 teramaster2
```

10.1.5 Input - File app_config.ini

Inside the application folder, there should be a `app_config.ini` file having the required specific ports for the application. If the application does not need any specific ports, then the `app_config.ini` with the two sections `[DOCKER_PORT]` and `[SVC_PORT]` should be left empty. The section `[SVC_PORT]` should specify the required ports needed for the application, and the section `[DOCKER_PORT]` should specify the corresponding target ports for the docker.

```
[DOCKER_PORT]
    PYTHON-PORT = 57021
[SVC_PORT]
    PYTHON-PORT = 57021
```

10.1.6 Input - File input_node.txt

This file is used by WAVE algorithm and provides the information of the compute node for the first task. Format of this file is given (in this case, node2 will perform the first task):

task	node
task0	node2

10.1.7 Input - File name_convert.txt

This file helps to output the correct performance statistics of Jupiter's runtime profiler, which lists all the task name, its corresponding input and output file name, line by line, in the following format:

input	output_app_name	input_app_name
task1	output_task1_name	input_task1_name
task2	output_task2_name	input_task2_name
task3	output_task3_name	input_task3_name

A given sample of `name_convert.txt` file:

10.1.8 Output

Note: Taking the node list from `nodes.txt` and DAG information from `configuration.txt`, Jupiter will consider both updated network connectivity (from `DRUPE-network profiler` or your chosen tool) and computational capabilities (from `DRUPE - resource profiler` or your chosen tool) of all the nodes in the system, Jupiter use the chosen scheduling algorithm (`HEFT original`, `random WAVE`,‘‘greedy WAVE‘‘ or `HEFT modified`) to give the optimized mapping of tasks and nodes in the system. Next, `CIRCE` will handle deploying the optimized mapping in the **Kubernetes** system.

10.1.9 Scripts format

Each task should be coded as a python script inside the `scripts` folder. Each code for a task must be placed inside a function called `task` that takes as arguments `(inputfiles, inputpath, outputpath)` and returns `a list of output files`.

```
input global botnet
localpro split botnet
aggregate0 merged split
aggregate1 merged split
aggregate2 merged split
simpledetector0 anomalies merged
simpledetector1 anomalies merged
simpledetector2 anomalies merged
astutedetector0 anomalies merged
astutedetector1 anomalies merged
astutedetector2 anomalies merged
fusioncenter0 fusion anomalies
fusioncenter1 fusion anomalies
fusioncenter2 fusion anomalies
globalfusion global fusion
```

A given example of a task:

In the above example, `onefile` parameter requires the list of input filenames to the tasks, `pathin` parameter requires the absolute path of the input folder, `pathout` parameter requires the absolute path of the output folder, and the `task` function returns the absolute path of the out files generated by the task itself.

10.2 Network Anomaly Detection

The Coded DNAD (Coded Network Anomaly Detection) is an application customized for Jupiter Orchestrator. Jupiter accepts pipelined computations described in a form of a Graph where the main task flow is represented as a Directed Acyclic Graph(DAG). Thus, one should be able separate the graph into two pieces, the DAG part and non-DAG part. Jupiter requires that each task in the DAG part of the graph to be written as a Python function in a separate file under the scripts folder. On the other hand the non-DAG tasks can be either Python function or a shell script with any number of arguments, located under the scripts folder.

10.2.1 Overview

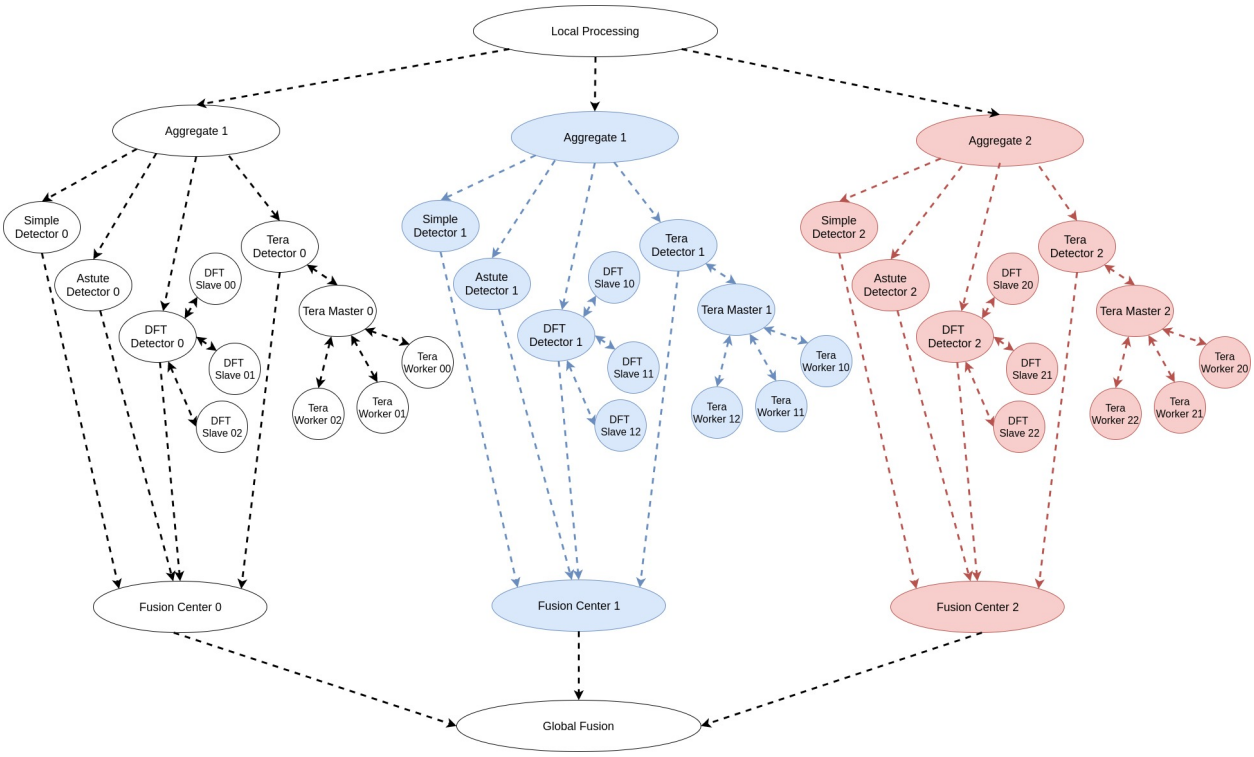
The application task graph, shown below, is intended for dispersed computing. It is inspired from Hashdoop [1, 2], where a MapReduce framework is used for anomaly detection. We have modified the codes from [2] to suit our purpose.

10.2.2 Input

Convert the pcap file to a text file using Ipsumdump as follows:

```
ipsumdump -tsSdDlpF -r botnet-capture-20110810-neris.pcap > botnet_summary.ipsum
```

```
import os
import time
import cv2 as cv

def task(onefile, pathin, pathout):


    filelist=[]
    filelist.append(onefile)

    #store the data&time info
    snapshot_time = filelist[0].partition('_')[2]
    time.sleep(10)



    for filename in filelist:

        # open the target jpeg and convert to gray scale
        src = cv.imread(os.path.join(pathin, filename))
        src = cv.cvtColor(src, cv.COLOR_BGR2GRAY)

        # Histogram Equalization to improve the contrast of image
        #dst = cv.equalizeHist(src)

        cv.imwrite(os.path.join(pathout,'processed1_'+snapshot_time), src)

    return [os.path.join(pathout,'processed1_'+snapshot_time)]



def main():
    filelist= 'camera1_20190222.jpeg'
    outpath = os.path.join(os.path.dirname(__file__), "generated_files/")
    outfile = task(filelist, outpath, outpath)
    return outfile

if __name__ == '__main__':

    #Suppose the file structure is erick/detection_app/camera1_input/camera1_20190222.jpeg
    filelist= 'camera1_20190222.jpeg'
    task(filelist, '/home/erick/detection_app/camera1_input', '/home/erick/detection_app')
```



10.2.3 References

[1] Romain Fontugne, Johan Mazel, and Kensuke Fukuda. “Hashdoop: A mapreduce framework for network anomaly detection.” Computer Communications Workshops (INFOCOM WORKSHOPS), IEEE Conference on. IEEE, 2014.

[2] Hashdoop GitHub Repository

[3] Fernando Silveira, Christophe Diot, Nina Taft, and Ramesh Govindan. “ASTUTE: Detecting a different class of traffic anomalies.” ACM SIGCOMM Computer Communication Review 40.4 (2010): 267-278.

For more information, please refer to README file of the Coded DNAD repo.

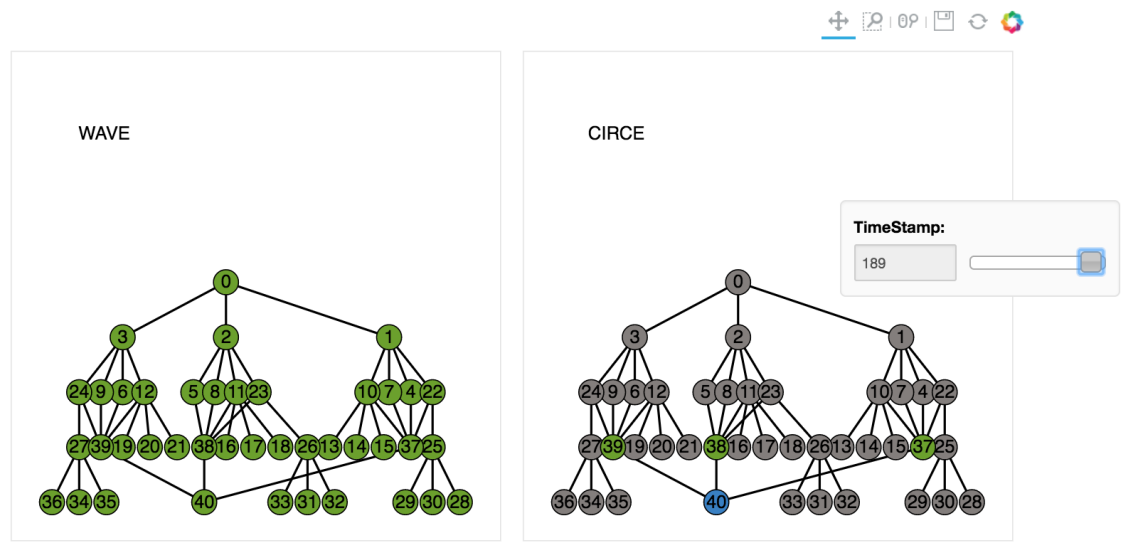
CHAPTER 11

Jupiter Visualization

The visualization tool for **Jupiter** is given here. This tool generates an interactive plot to show the scheduling result of WAVE and the dispatcher mapping of CIRCE.

```
WARNING:root:Warning: Nesting Layouts within a HoloMap makes it difficult to access your data or control how it appea
rs; we recommend calling .collate() on the HoloMap in order to follow the recommended nesting structure shown in the
Composing Data tutorial(https://goo.gl/2YS8LJ)
```

Out[4]:



Warning: To visualize your own application, make sure the format of your logs are in line with the input files of the tools. We will integrate this as a real-time visualization tool for Jupiter in the next release.

CHAPTER 12

Troubleshooting

Warning: You will notice that some times the jupiter deploy script says that certain pods are not running. If some pods persist in the list for a long time, it is advisable to look into the k8 dashboard to check whether the pod is on a CrashLoopBack state. If yes, you can just delete the pod using the dropdown menu in the dashboard. **DO NOT Delete Anything but the pod i.e. DO NOT delete the SVC, Deployment or the Recplicaset**. By doing this, the k8 will spawn a new instance of the pod. If the problem persists, look at the logs or the error message for more details.

CHAPTER 13

Acknowledgement

This material is based upon work supported by **Defense Advanced Research Projects Agency (DARPA)** under Contract No. **HR001117C0053**. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

CHAPTER 14

LICENSE

Developed by:

- **Autonomous Networks Research Group** (ANRG)
- **University of Southern California**

Contributors:

- Quynh Nguyen
- Pradipta Ghosh
- Pranav Sakulkar
- Aleksandra Knezevic
- Jiatong Wang
- Jason Tran
- Bhaskar Krishnamachari

- A citation to the Autonomous Networks Research Group must be included in any publications benefiting from the use of the Software.

THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

CHAPTER 15

Overview

DRUPE is a tool to collect information about computational resources as well as network links between compute nodes in a dispersed computing system to a central node. DRUPE consists of a network profiler and a resource profiler.

Note: We can use DRUPE as an independent tool.

The code is open source, and available on GitHub.

15.1 Network Profiler

15.1.1 Description

The network profiler in DRUPE automatically schedules and logs communication information of all links between nodes in the network, which gives the quadratic regression parameters of each link representing the corresponding communication cost. The quadratic function represents how the file transfer time depends on the file size (based on our empirical finding that a quadratic function is a good fit.)

15.1.2 Versions

- Non dockerized implementation
- Dockerized implementation
- Kubernetes implementation

15.1.3 Input

- File `central.txt` stores credential information of the central node

CENTRAL IP	USERNAME	PASSWORD
IP0	USERNAME	PASSWORD

- File `nodes.txt` stores credential information of the nodes information

TAG	NODE (username@IP)	REGION
node1	username@IP1	LOC1
node2	username@IP2	LOC2
node3	username@IP3	LOC3

- File `link_list.txt` stores the the links between nodes required to log the communication

SOURCE(TAG)	DESTINATION(TAG)
node1	node2
node1	node3
node2	node1
node2	node3
node3	node1
node3	node2

- File `generate_link_list.py` is used to generate file `link_list.txt` (all combinations of links) from the node list in file `nodes.txt`, or users can customize the `link_list.txt` on their own.

15.1.4 Output

All quadratic regression parameters are stored in the local MongoDB server on the central node.

15.2 Resource Profiler

15.2.1 Description

This Resource Profiler will get system utilization from all the nodes in the system. These information will then be sent to home node and stored into mongoDB.

15.2.2 Output

The information includes: IP address of each node, cpu utilization of each node, memory utilization of each node, and the latest update time.

CHAPTER 16

How to run

16.1 Network Profiler

16.1.1 Non-dockerized version

- At the central network profiler:
  - Install required libraries:

```
./central_init
```

  - Inside the folder central, input add information about the nodes and the links.
  - Generate the scheduling files for each node, prepare the central database and collection, copy the scheduling information and network scripts for each node in the node list and schedule updating the central database every 10th minute.

```
python3 central scheduler.py
```

- At the droplets:
  - The central network profiler copies all required scheduling files and network scripts to the folder online profiler in each droplet.
  - Install required libraries

```
./droplet_init
```

  - Generate files with different sizes to prepare for the logging measurements, generate the droplet database, schedule logging measurement every minute and logging regression every 10th minute. (These parameters could be changed as needed.)

```
python3 automate droplet.py
```

16.1.2 Dockerized version

- At the docker_online_profiler folder:
  - Modify input in folder `central_input` (`nodes.txt`, `link_list.txt`) of central_network_profiler and upload_docker_network accordingly (IP, PASSWORD, REG, link_list)
  - Upload codes to all the nodes and the central

```
./upload_docker_network
```

  - Example run: Scheduler IP0, and other droplets IP1, IP2, IP3
- At the droplets, inside the droplet_network_profiler:
  - Build the docker:

```
docker build -t droplet_network_profiler .
```

  - Run the containers:

```
docker run --rm --name droplet_network_profiler -t -i -e DOCKER_HOST=IP1 -p 5100:22 -P droplet_network_profiler

docker run --rm --name droplet_network_profiler -t -i -e DOCKER_HOST=IP2 -p 5100:22 -P droplet_network_profiler

docker run --rm --name droplet_network_profiler -t -i -e DOCKER_HOST=IP3 -p 5100:22 -P droplet_network_profiler
```

- At the central network profiler (IP0):
  - Build the docker:

```
docker build -t central_network_profiler .
```

  - Run the container:

```
docker run --rm --name  central_network_profiler -i -t -e DOCKER_HOST=IP0 -p 5100:22 -P central_network_profiler
```

16.1.3 Kubernetes Version of Network Profiler

Run from scratch

- The instructions here begin at the point in which you have a target `configuration.txt` and `nodes.txt` file. First, you need to build your Docker images. There are currently two separate images: the `central_profiler` image and `worker_profiler` image.
- To rebuild Docker images and push them to the ANRG Docker Hub repo, first login to Docker Hub using your own credentials by running `docker login`. Then, in the folder with the Dockerfile files, use this template to build all the needed Docker images:

```
docker build -f $target_dockerfile . -t $dockerhub_user/$repo_name:$tag
docker push $dockerhub_user/$repo_name:$tag
```

- Example:

```
docker build -f Network_Profiler/central_network_profiler/Dockerfile . -t anrg/
↪central_profiler:v1
docker push anrg/central_profiler:v1
docker build -f Network_Profiler/droplet_network_profiler/Dockerfile . -t anrg/worker_
↪profiler:v1
docker push anrg/worker_profiler:v1
```

- Note: If you just want to control the whole cluster via our master node (i.e. you don't want to use your computer) go to *this section* in the readme).
- To control the cluster, you need to grab the `admin.conf` file from the k8s master node. When the cluster is bootstrapped by `kubeadm` see the k8s cluster setup notes here the `admin.conf` file is stored in `/etc/kubernetes/admin.conf`. Usually, a copy is made into the `$HOME` folder. Either way, make a copy of `admin.conf` into your local machine's home folder. Then, make sure you have `kubectl` installed instructions here.
- Next, you need to run the commands below. You can wrap it up in a script you source or directly place the export line and source line into your .bashrc file. However, make sure to re-run the full set of commands if the `admin.conf` file has changed:

```
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf #check if it works with `kubectl get nodes`
source <(kubectl completion bash)
```

- Clone or pull this repo and `cd` into the repo's directory. Currently, you need to have `admin.conf` in the folder above your clone. Our python scripts need it exactly there to work. Then, run:

```
python3 k8s_profiler_scheduler.py
```

Then wait for a bit like 2-3 min for all the worker dockers to be up and running. Then run:

```
python3 k8s_profiler_home_scheduler.py
```

- Lastly, you will want to access the k8s Web UI on your local machine. Assuming you have `kubectl` installed and `admin.conf` imported, simply open a separate terminal on your local machine and run:

```
kubectl proxy
```

- The output should be something like:

```
Starting to serve on 127.0.0.1:8001
```

- Open up a browser on your local machine and go to `http://127.0.0.1:8001/ui`. You should see the k8s dashboard. Hit `Ctrl+c` on the terminal running the server to turn off the proxy. Alternatively, you can run this command directly in the folder where the `admin.conf` file is (not recommended):

```
kubectl --kubeconfig=./admin.conf proxy - p 80
```

Teardown

- To teardown the DAG deployment, run the following:

```
python3 delete_all_profilers.py
```

- Once the deployment is torn down, you can simply start from the begining of these instructions to make changes to your code and redeploy the DAG. FYI, k8s_scheduler.py defaults to ALWAYS pulling the Docker image (even if it hasn't changed).

Controlling Cluster from K8s Master Node

- Login to the Kubernetes Master node (currently Jason's computer under the user `apac`). Assuming the cluster is up (it typically will not be shutdown), source the sourceit.sh script in the `apac` user's home folder so you can use `kubectl` to control the cluster:

```
source sourceit.sh
```

- Note that you do NOT need to do this if the `admin.conf` file hasn't changed given the following lines are placed in the master node's .bashrc file:

```
export KUBECONFIG=$HOME/admin.conf
source <(kubectl completion bash)
```

The `admin.conf` file changes whenever the cluster is re-bootstrapped. You can then run the following command to check if everything is working. If it lists all the nodes in the cluster, you're ready to start controlling it:

```
kubectl get nodes #if this works you're ready to start Controlling
```

16.2 Resource Profiler

16.2.1 Non-dockerized version

- For working nodes:
  - copy the `Resource_Profiler_server` folder to each working node using scp.
  - In each node:

```
python2 Resource_Profiler_server/install_package.py
```

- For scheduler node:
  - copy `Resource_Profiler_control` folder to home node using scp.
  - if a node's IP address changes, just update the `Resource_Profiler_control/ip_path` file
  - optional: inside `Resource_Profiler_control` folder:

```
1 python2 install_package.py
2 python2 jobs.py &
```

- Note: the content of ip_path are several lines of working nodes' IP address. So if a node's IP address is changed, make sure to update the ip_path file.

16.2.2 Dockerized-version

- For working nodes:
  - copy the `Resource_Profiler_server_docker` folder to each working node using scp.

  - in each node:

```
docker build -t server .
docker run -d -p 49155:5000 server
```

- For scheduler node:
  - copy `Resource_Profiler_control_docker` folder to home node using scp.
  - if a node's IP address changes, just update the `Resource_Profiler_control_docker/control_file/ip_path` file
  - optional: find central_network_profiler container Get the IP address.

```
docker inspect CONTAINER ID
```

  - type mongo IP and then inside mongo shell.

```
use DBNAME db.createUser({ user: 'USERNAME', pwd: 'PASSWORD', roles: [{␣
↪role: 'readWrite', db:'DBNAME'}] }}
```

  - inside `Resource_Profiler_control_docker` folder:

```
docker build -t control . docker run control
```

- Note: the content of ip_path are several lines of working nodes' IP address. So if a node's IP address is changed, make sure to update the ip_path file.

CHAPTER 17

Overview

WAVE is a distributed scheduler for DAG type task graph that outputs a mapping of tasks to real compute nodes. It is a module used in Jupiter. We have two versions random_WAVE and greedy_WAVE

Note: We can not use WAVE as an independent tool

The code is open source, and available on GitHub.

17.1 Description

Given a task graph that is represented as `directed acyclic graph` (DAG) and a network of `network compute points` (NCPs), the dispersed computing scheduler needs to figure out a mapping from the tasks to the NCPs with the goal **minimizing the average end-to-end latency of the incoming data-frames**. The scheduler needs to know the **computing resources availability** at each of the NCPs and the **qualities of the links** connecting any two devices in order to come up with a mapping.

- The knowledge of the compute resources is important, since different tasks might be executed on different NCPs.
- In such a case, the link quality knowledge is especially important as the output of a task being executed at an NCP will need to be shipped to its children task being executed at another NCP.

We consider the case where the scheduling is done in a distributed manner by multiple collaborating nodes located at different geographical locations. In such cases, each scheduling node only needs to know about its neighbours, their compute profiles and the link qualities to them.

17.2 WAVE components

Our scheduler is initialized by the WAVE master node, which has the information about the task graph. Based on the location of the input data, it determines the NCPs for each of the input tasks in the DAG. Also the master node

determines a unique parent controller for each task in te task graph using following routine 1 shown below. At the WAVE master node, following routine is executed to determine the controllers for each task.

Routine 1: Controller section routine

- Iterate over tasks of the the task graph in their topological orders.
- For each non-input task, check if any of its parent tasks are already controllers.
- If only one of the parents is already a controller, then appoint that parent as the controller for this task.
- If no parent is already a controller, then choose the task with smaller topological index as the parent.

The task graph, input NCPs and parent controllers for each task are then sent to all NCPs. All NCPs are waiting to receive their task assignments. Whenever an NCP hears its task responsibility, it first needs to check if the assigned task is also a controller for any of the other tasks. The NCP can check this by looking up the parent controller information it has received from the WAVE master. If the NCP is a controller parent for some tasks of the task graph, it needs to do a NCP assignment for the children tasks. It runs following routine 2 to perform the child appointment.

Routine 2: Scheduling algorithm at the controller

- Iterate over the children tasks in their topological orders.
- For each task, randomly select an NCP from the neighbouring nodes.
- Convey the task appointments to the selected NCPs and the WAVE master node.

When the WAVE master hears about all the task assignments, it starts the CIRCE deployment framework by providing the task graph and the mapping of these tasks to the compute nodes.

Note: Step 2 in routine 2 selects the NCPs randomly, however our system implementation is very modular and we are working on replacing it with a child appointment algorithm that considers the execution profiles of the neighbours, the link qualities connecting them and also the computing and communication requirements of the concerned tasks.

17.3 WAVE scheduling algorithms

17.3.1 Random WAVE

When assigning tasks to nodes, this version of **WAVE** does random assignment. The algorithm will random choose a node in the set of all validate nodes.

17.3.2 Greedy WAVE

When assigning tasks to nodes, this version of **WAVE** will first get network parameters from DRUPE as a delay factor. The delay factor contains network delay, target node's CPU usage and memory usage. The **WAVE** running in each of the node will maintain a sorted list of all its neighbor's delay information. Then it will pick up the node who has the lowest delay and assign tasks to it.

CHAPTER 18

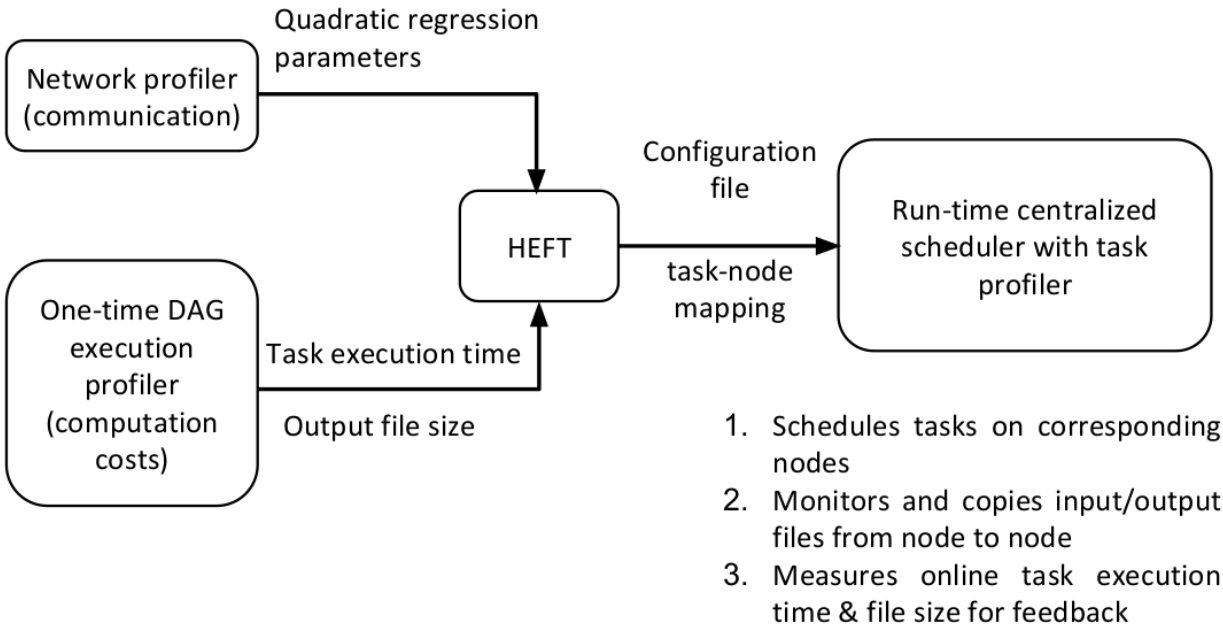
How to run

You need to have Jupiter first. Then copy either `random_WAVE` folder or `greedy_WAVE` folder to Jupiter's root folder, and rename as `wave`. Then it's good to use.

CHAPTER 19

Overview

CIRCE is a runtime scheduling software tool for dispersed computing, which can deploy pipelined computations described in the form of a directed acyclic graph (DAG) on multiple geographically dispersed computers (compute nodes).



The tool run on a host node (also called scheduler node). It needs the information about compute nodes are available (such as IP address, username and password), the description of the DAG along with code for the corresponding tasks. Based on measurements of computation costs for each task on each node and the communication cost of transferring output data from one node to another, it first uses a DAG-based static scheduling algorithm (at present, we include a modified version of an implementation [2] of the well-known HEFT algorithm [1] with the tool) to determine at which node to place each task from the DAG. CIRCE then deploys the each tasks on the corresponding node, using input and

output queues for pipelined execution and taking care of the data transfer between different nodes.

Note: We can use CIRCE as an independent tool.

The code is open source, and available on GitHub.

We also use Distributed Network Anomaly Detection application (DNAD) as an application example.

19.1 Inputs

19.1.1 File nodes.txt

- List of nodes for the experiment, including the scheduler node. The user needs to fill the file with the appropriate IP addresses, usernames and passwords of their compute nodes:

scheduler	IP	username	pw
node1	IP	username	pw
node2	IP	username	pw
node3	IP	username	pw

19.1.2 File dag.txt and file configuration.txt

- DAG description of DAND (as adjacency list, where the first item is a parent task and subsequent items are child tasks) is kept in two files `dag.txt` (used by HEFT) and `configuration.txt` (`config_security.txt` in this example):
  - Format of `dag.txt`:

```
local_pro aggregate0 aggregate1 aggregate2
aggregate0 simple_detector0 astute_detector0
aggregate1 simple_detector1 astute_detector1
aggregate2 1 true simple_detector2 astute_detector2
simple_detector0 fusion_center0
simple_detector1 fusion_center1
simple_detector2 fusion_center2
astute_detector0 fusion_center0
astute_detector1 fusion_center1
astute_detector2 fusion_center2
fusion_center0 global_fusion
fusion_center1 global_fusion
fusion_center2 global_fusion
global_fusion scheduler
```

  - Format of `configuration.txt`:

```
14
local_pro 1 false aggregate0 aggregate1 aggregate2
aggregate0 1 true simple_detector0 astute_detector0
aggregate1 1 true simple_detector1 astute_detector1
aggregate2 1 true simple_detector2 astute_detector2
simple_detector0 1 true fusion_center0
simple_detector1 1 true fusion_center1
```

(continues on next page)

(continued from previous page)

```
simple_detector2 1 true fusion_center2
astute_detector0 1 true fusion_center0
astute_detector1 1 true fusion_center1
astute_detector2 1 true fusion_center2
fusion_center0 2 true global_fusion
fusion_center1 2 true global_fusion
fusion_center2 2 true global_fusion
global_fusion 3 true scheduler
```

- The first line is an integer, which gives the number of lines the DAG is taking in the file. DAG is represented in the form of adjacency list:

```
parent_task NUM_INPUTS FLAG child_task1 child_task2 child task3 ...
```

- `parent_task` is the name of the parent task
- `NUM_INPUTS` is an integer representing the number of input files the task needs in order to start processing (some tasks could require more than input).
- `FLAG` is `true` or `false`. Based on its value, `monitor.py` will either send a single output of the task to all its children (when true), or it will wait the output files and start putting them into queue (when false). Once the queue size is equal to the number of children, it will send one output to one child (first output to first listed child, etc.).
- `child_task1`, `child_task2`, `child_task3...` are the names of child tasks of the current parent task.

CHAPTER 20

Userguide

20.1 Profiling

20.1.1 Execution profiler

- Description: produces `profiler_nodeX.txt` file for each node, which gives the execution time of each task on that node and the amount of data it passes to its child tasks. These results are required in the next step for HEFT algorithm.
- Input: `dag.txt`, `nodes.txt`, DAG task files (`task1.py`, `task2.py`,...), DAG input file (`input.txt`)
- Output: `profiler_nodeNUM.txt`
- How to run
  - Case 1: the file `scheduler.py` will copy the `app` folder to each of the nodes and execute the docker commands. Inside `circe/docker_execution_profiler` folder perform the following command:

```
python3 scheduler.py
```

  - Case 2: copy the `app` folder to the each of the nodes using scp and inside `app` folder perform the following commands where hostname is the name of the node (node1, node2, etc.).

```
docker build -t profilerimage . docker run -h hostname profilerimage
```

  - In both cases make sure that the command inside file `app/start.sh` gives the details (IP, username and password) of your scheduler machine.

20.1.2 Central network profiler

- Description: automatically scheduling and logs communication information of all links betweet nodes in the network, which gives the quaratic regression parameters of each link representing the corresponding communication cost. These results are required in the next step for HEFT algorithm.

- Input:
  - File `central.txt` stores credential information of the central node

CENTRAL IP	USERNAME	PASSWORD
IP0	USERNAME	PASSWORD

  - File `nodes.txt` stores credential information of the nodes information

TAG	NODE (username@IP)	REGION
node1	username@IP1	LOC1
node2	username@IP2	LOC2
node3	username@IP3	LOC3

  - File `link_list.txt` stores the the links between nodes required to log the communication

SOURCE(TAG)	DESTINATION(TAG)
node1	node2
node1	node3
node2	node1
node2	node3
node3	node1
node3	node2

- Output: all quadratic regression parameters are stored in the local MongoDB on the central node.
- **How to run:**
  - At the central network profiler:
    - Install required libraries:

```
./central_init
```

    - Inside the folder central, input add information about the nodes and the links.
    - Generate the scheduling files for each node, prepare the central database and collection, copy the scheduling information and network scripts for each node in the node list and schedule updating the central database every 10th minute.

```
python3 central scheduler.py
```

  - At the droplets:
    - The central network profiler copies all required scheduling files and network scripts to the folder online profiler in each droplet.
    - Install required libraries

```
./droplet_init
```

    - Generate files with different sizes to prepare for the logging measurements, generate the droplet database, schedule logging measurement every minute and logging regression every 10th minute. (These parameters could be changed as needed.)

```
python3 automate droplet.py
```

20.1.3 System resource profiler

- Description: This Resource Profiler will get system utilization from all the nodes in the system. These information will then be sent to home node and stored into mongoDB.
- Output: The information includes: IP address of each node, cpu utilization of each node, memory utilization of each node, and the latest update time.
- How to run:
  - For working nodes:
    - copy the `Resource_Profiler_server` folder to each working node using scp.
    - In each node:

```
python2 Resource_Profiler_server/install_package.py
```

  - For scheduler node:
    - copy `Resource_Profiler_control` folder to home node using scp.
    - if a node's IP address changes, just update the `Resource_Profiler_control/ip_path` file
    - optional: inside `Resource_Profiler_control` folder:

```
1 python2 install_package.py
2 python2 jobs.py &
```

  - Note: the content of `ip_path` are several lines of working nodes' IP address. So if a node's IP address is changed, make sure to update the `ip_path` file.

20.2 Heft

- Description: This HEFT implementation has been adapted/modified from [2].
- Input: HEFT implementation takes a file of .tgff format, which describes the DAG and its various costs, as input. The first step is to construct this (`input.tgff`) file from the input files `dag.txt`, `profiler_nodeNUM.txt`. From `circe/heft/` folder execute:

```
python write_input_file.py
```

- HEFT algorithm: This is the scheduling algorithm which decides where to run each task. It writes its output in a configuration file, needed in the next step by the run-time centralized scheduler. The algorithm takes input.tgff as an input and output the scheduling file `configuration.txt`. From `circe/heft/` run:

```
python main.py
```

20.3 Centralized scheduler with profiler

- Centralized run-time scheduler. This is the run-time scheduler. It takes the configuration file `configuration.txt`, given by HEFT, the node information `nodes.txt` and orchestrates the execution of tasks on given nodes, and output the DAG output files in `circe/centralized_scheduler/output/` folder. Inside `circe/centralized_scheduler` folder run:

```
python3 scheduler.py
```

- Wait several seconds and move `input1.txt` to `apac_scheduler/centralized_scheduler/input/` folder (repeat the same for other input files).
- Stopping the centralized run-time scheduler. Run:

```
python3 removeprocesses.py
```

  This script will shh into every node and kill running processes, and kill the process on the master node.

- If network conditions change, one might want to restart the whole application. This can be done by running:

```
python3 remove_and_restart.py
```

  The first part of the script stops the system as described above. It then runs HEFT and restarts the centralized run-time scheduler with the new task-node mapping.

20.4 Run-time task profiler

CHAPTER 21

Project Structure

It is assumed that the folder `circe/` is located on the users home path (for example: `/home/apac`). The structure of the project within `circe/` folder is the following:

```
- nodes.txt
- dag.txt
- configuration.txt (output of the HEFT algorithm)
- profiler node1.txt, profiler node2.txt,... (output of execution profiler)
- docker_execution_profiler/
    - scheduler.py
    - app/
        - dag.txt
        - requirements.txt
        - Dockerfile
        - DAG task files (task1.py, task2.py,...)
        - DAG input file (input1.txt)
        - start.sh
        - profiler.py
- centralized scheduler with profiler/
    - input/ (this folder should be created by user)
    - output/ (this folder should be created by user)
    - 1botnet.ipsum, 2botnet.ipsum (example input files)
    - scheduler.py
    - monitor.py
    - securityapp (this folder contains application task files, in this case localpro.
↪py, aggregate0.py,...)
    - removeprocesses.py
    - remove_and_restart.py
    - readconfig.py
- heft/
    - write_input_file.py
    - heft_dup.py
    - main.py
    - create_input.py
    - cpop.py
```

(continues on next page)

(continued from previous page)

```
    - read config.py
    - input.tgff (output of write input file.py)
    - readme.md
- central network profiler/
    - folder central_input: link list.txt, nodes.txt, central.txt
    - central copy nodes
    - central init
    - central query statistics.py
    - central scheduler.py
    - folder network script: automate droplet.py, droplet generate random files,␣
↪droplet init, droplet scp time transfer
    - folder mongo control
        - mongo scrip/
          - server.py
          - install package.py
        - mongo control/
          - insert to mongo.py
          - read info.py
          - read info.pyc
          - install package.py
          - jobs.py
          - ip path
```

Note that while we currently use an implementation of HEFT for use with CIRCE, other schedulers may be used as well.

CHAPTER 22

References

[1] H. Topcuoglu, S. Hariri, M.Y. Wu, **Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing**, IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 3, pp. 260 - 274, 2002.

[2] Ouyang Liduo, **HEFT Implementation Original Source Code**, source link (we have modified this code in ours.)

CHAPTER 23

Circe Reference

Note: This page gives an overview of all public **CIRCE** objects, functions and methods which have been used in the **JUPITER** system.

23.1 Original CIRCE (Non-pricing CIRCE)

23.2 Pricing CIRCE (Event driven scheme)

23.3 Pricing CIRCE (Push scheme)

CHAPTER 24

Profilers Reference

Note: This page gives an overview of all public profilers objects (network profiler, resource profiler and execution profiler), functions and methods which have been used the **JUPITER** system.

24.1 Network Profiler

24.2 Resource Profiler

24.3 Execution Profiler

CHAPTER 25

Task Mapper Reference

Note: This page gives an overview of all public Task Mapper objects (including **HEFT** and **WAVE** scheduling algorithm), functions and methods which have been used the **JUPITER** system.

25.1 HEFT

25.2 WAVE

25.2.1 Random Wave

25.2.2 Greedy Wave

CHAPTER 26

Scripts Reference

Note: This page gives an overview of all public **Scripts** which helps to build and deploy the **JUPITER** system in **Kubernetes**.

26.1 Build scripts

26.2 Teardown scripts

26.3 Deploy scripts

26.4 Configuration scripts

26.5 Docker file preparation scripts

26.6 Other scripts

CHAPTER 27

Jupiter Indices

- modindex