maro

MARO Team

Jul 05, 2022

INSTALLATION

Multi-Agent Resource Optimization (MARO) platform is an instance of Reinforcement learning as a Service (RaaS) for real-world resource optimization. It can be applied to many important industrial domains, such as container inventory management in logistics, bike repositioning in transportation, virtual machine provisioning in data centers, and asset management in finance. Besides Reinforcement Learning (RL), it also supports other planning/decision mechanisms, such as Operations Research.

CHAPTER

ONE

KEY COMPONENTS

- Simulation toolkit: it provides some predefined scenarios, and the reusable wheels for building new scenarios.
- RL toolkit: it provides a full-stack abstraction for RL, such as agent manager, agent, RL algorithms, learner, actor, and various shapers.
- Distributed toolkit: it provides distributed communication components, interface of user-defined functions for message auto-handling, cluster provision, and job orchestration.

CHAPTER

TWO

QUICK START

```python
from maro.simulator import Env
from maro.simulator.scenarios.cim.common import Action, ActionType, DecisionEvent

from random import randint

# Initialize an Env for cim scenario
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0", start_tick=0, durations=100)

metrics: object = None
decision_event: DecisionEvent = None
is_done: bool = False
action: Action = None

# Start the env with a None Action
metrics, decision_event, is_done = env.step(None)

while not is_done:
    # Generate a random Action according to the action_scope in DecisionEvent
    action_scope = decision_event.action_scope
    to_discharge = action_scope.discharge > 0 and randint(0, 1) > 0

    action = Action(
        decision_event.vessel_idx,
        decision_event.port_idx,
        randint(0, action_scope.discharge if to_discharge else action_scope.load),
        ActionType.DISCHARGE if to_discharge else ActionType.LOAD
    )

    # Respond the environment with the generated Action
    metrics, decision_event, is_done = env.step(action)
```

CHAPTER

THREE

CONTENTS

3.1 Package

3.1.1 Install MARO from PyPI

- Max OS / Linux

```
pip install pymaro
```

- Windows

```
# Install torch first, if you don't have one.
pip install torch===1.6.0 torchvision===0.7.0 -f https://download.pytorch.org/whl/
↪torch_stable.html

pip install pymaro
```

3.1.2 Install MARO from Source (Editable Mode)

- Prerequisites
  - Python >= 3.7
  - C++ Compiler
    * Linux or Mac OS X: `gcc`
    * Windows: Build Tools for Visual Studio 2017
- Enable Virtual Environment
  - Mac OS / Linux

    ```
    # If your environment is not clean, create a virtual environment firstly.
    python -m venv maro_venv
    source ./maro_venv/bin/activate
    ```

  - Windows

    ```
    # If your environment is not clean, create a virtual environment firstly.
    python -m venv maro_venv
    .\maro_venv\Scripts\activate
    ```

- Install MARO

- Mac OS / Linux

```
# Install MARO from source.
bash scripts/install_maro.sh
```

- Windows

```
# Install MARO from source.
.\scripts\install_maro.bat
```

3.2 Playground Docker Image

3.2.1 Pull from Docker Hub

```
# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Jupyter lab with maro -> http://127.0.0.1:40010
docker run -p 40009:40009 -p 40010:40010 maro2020/playground
```

3.2.2 Run from Source

- Mac OS / Linux

```
# Build playground image.
bash ./scripts/build_playground.sh

# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Jupyter lab with maro -> http://127.0.0.1:40010
docker run -p 40009:40009 -p 40010:40010 maro2020/playground
```

- Windows

```
# Build playground image.
.\scripts\build_playground.bat

# Run playground container.
# Redis commander (GUI for redis) -> http://127.0.0.1:40009
# Jupyter lab with maro -> http://127.0.0.1:40010
docker run -p 40009:40009 -p 40010:40010 maro2020/playground
```

3.2.3 Major Services in Playground

Service	Description	Host
`Redis Commander`	Redis web GUI.	http://127.0.0.1:40009
`Jupyter Lab`	Jupyter lab with MARO environment, examples, notebooks.	http://127.0.0.1:40010

(Remember to change ports if you use different ports mapping.)

3.2.4 Major Materials in Root Folder

Folder	Description
`examples`	Showcases of predefined scenarios.
`notebooks`	Quick-start tutorial.

(The ones not mentioned in this table can be ignored.)

3.3 Grass Cluster Provisioning on Azure

With the following guide, you can build up a MARO cluster in *grass/azure* mode on Azure and run your training job in a distributed environment.

3.3.1 Prerequisites

- Install the Azure CLI (preferred version: v2.20.0) and login
- Install docker and Configure docker to make sure it can be managed as a non-root user

3.3.2 Cluster Management

- Create a cluster with a deployment

```
# Create a grass cluster with a grass-create deployment
maro grass create ./grass-azure-create.yml
```

- Scale the cluster

  Check VM Size to see more node specifications.

```
# Scale nodes with 'Standard_D4s_v3' specification to 2
maro grass node scale myGrassCluster Standard_D4s_v3 2

# Scale nodes with 'Standard_D2s_v3' specification to 0
maro grass node scale myGrassCluster Standard_D2s_v3 0
```

- Delete the cluster

```
# Delete a grass cluster
maro grass delete myGrassCluster
```

- Start/Stop nodes to save costs

```
# Start 2 nodes with 'Standard_D4s_v3' specification
maro grass node start myGrassCluster Standard_D4s_v3 2

# Stop 2 nodes with 'Standard_D4s_v3' specification
maro grass node stop myGrassCluster Standard_D4s_v3 2
```

- Get statuses of the cluster

```
# Get master status
maro grass status myGrassCluster master

# Get nodes status
maro grass status myGrassCluster nodes

# Get containers status
maro grass status myGrassCluster containers
```

- Clean up the cluster

  Delete all running jobs, schedules, containers in the cluster.

```
maro grass clean myGrassCluster
```

3.3.3 Run Job

- Push your training image from local machine

```
# Push image 'myImage' to the cluster,
# 'myImage' is a docker image that loaded on the machine that executed this␣
↪command
maro grass image push myGrassCluster --image-name myImage
```

- Push your training data

```
# Push dqn folder under './myTrainingData/' to a relative path '/myTrainingData'␣
↪in the cluster
# You can then assign your mapping location in the start-job-deployment
maro grass data push myGrassCluster ./myTrainingData/dqn /myTrainingData
```

- Start a training job with a *start-job-deployment*

```
# Start a training job with a start-job deployment
maro grass job start myGrassCluster ./grass-start-job.yml
```

- Or, schedule batch jobs with a *start-schedule-deployment*

  These jobs will shared the same specification of components.

  A best practice to use this command will be: Push your training configs all at once with "`maro grass data push`", and get the jobName from environment variables in the containers, then use the specific training config based on the jobName.

```
# Start a training schedule with a start-schedule deployment
maro grass schedule start myGrassCluster ./grass-start-schedule.yml
```

- Get the logs of the job

```
# Get the logs of the job
maro grass job logs myGrassCluster myJob1
```

- List the current status of the job

```
# List the current status of the job
maro grass job list myGrassCluster
```

- Stop a training job

```
# Stop a training job
maro grass job stop myJob1
```

3.3.4 Sample Deployments

grass-azure-create

```
mode: grass/azure
name: myGrassCluster

cloud:
  resource_group: myResourceGroup
  subscription: mySubscription
  location: eastus
  default_username: admin
  default_public_key: "{ssh public key}"

user:
  admin_id: admin

master:
  node_size: Standard_D2s_v3
```

grass-start-job

You can replace {project root} with a valid linux path. e.g. /home/admin

Then the data you push will be mount into this folder.

```
mode: grass
name: myJob1

allocation:
  mode: single-metric-balanced
  metric: cpu

components:
  actor:
    command: "python {project root}/myTrainingData/dqn/job1/start_actor.py"
    image: myImage
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 1
      gpu: 0
      memory: 1024m
  learner:
    command: "python {project root}/myTrainingData/dqn/job1/start_learner.py"
    image: myImage
    mount:
      target: "{project root}"
```

(continues on next page)

(continued from previous page)

```
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

grass-start-schedule

```
mode: grass
name: mySchedule1

allocation:
  mode: single-metric-balanced
  metric: cpu

job_names:
  - myJob2
  - myJob3
  - myJob4
  - myJob5

components:
  actor:
    command: "python {project root}/myTrainingData/dqn/schedule1/actor.py"
    image: myImage
    mount:
      target: “{project root}”
    num: 5
    resources:
      cpu: 1
      gpu: 0
      memory: 1024m
  learner:
    command: "bash {project root}/myTrainingData/dqn/schedule1/learner.py"
    image: myImage
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048m
```

3.4 Grass Cluster Provisioning in On-Premises Environment

With the following guide, you can build up a MARO cluster in *grass/on-premises* in local private network and run your training job in On-Premises distributed environment.

3.4.1 Prerequisites

- Linux with Python 3.7+
- Install Powershell if you are using Windows Server
- For master node vm, need install flask, gunicorn, and redis.

3.4.2 Cluster Management

- Create a cluster with a *deployment*

```
# Create a grass cluster with a grass-create deployment
maro grass create ./grass-azure-create.yml
```

- Let a node join a specified cluster

```
# Let a worker node join into specified cluster
maro grass node join ./node-join.yml
```

- Let a node leave a specified cluster

```
# Let a worker node leave a specified cluster
maro grass node leave {cluster_name} {node_name}
```

- Delete the cluster

```
# Delete a grass cluster
maro grass delete my_grass_cluster
```

3.4.3 Run Job

See *Run Job in grass/azure* for reference.

3.4.4 Sample Deployments

grass-on-premises-create

```
mode: grass/on-premises
name: clusterName

user:
  admin_id: admin

master:
  username: root
  hostname: maroMaster
  public_ip_address: 137.128.0.1
  private_ip_address: 10.0.0.4
```

grass-on-premises-join-cluster

```
mode: grass/on-premises

master:
  private_ip_address: 10.0.0.4

node:
  hostname: maroNode1
  username: root
  public_ip_address: 137.128.0.2
  private_ip_address: 10.0.0.5
  resources:
    cpu: all
    memory: 2048m
    gpu: 0

 config:
   install_node_runtime: true
   install_node_gpu_support: false
```

3.5 K8S Cluster Provisioning on Azure

With the following guide, you can build up a MARO cluster in *k8s/aks* on Azure and run your training job in a distributed environment.

3.5.1 Prerequisites

- Install the Azure CLI and login
- Install and set up kubectl
- Install docker and Configure docker to make sure it can be managed as a non-root user
- Download AzCopy, then move the AzCopy executable to /bin folder or add the directory location of the AzCopy executable to your system path:

```
# Take AzCopy version 10.6.0 as an example

# Linux
tar xvf ./azcopy_linux_amd64_10.6.0.tar.gz; cp ./azcopy_linux_amd64_10.6.0/azcopy /
↪usr/local/bin

# MacOS (may required MacOS Security & Privacy setting)
unzip ./azcopy_darwin_amd64_10.6.0.zip; cp ./azcopy_darwin_amd64_10.6.0/azcopy /usr/
↪local/bin

# Windows
# 1. Unzip ./azcopy_windows_amd64_10.6.0.zip
# 2. Add the path of ./azcopy_windows_amd64_10.6.0 folder to your Environment␣
↪Variables
# Ref: https://superuser.com/questions/949560/how-do-i-set-system-environment-
↪variables-in-windows-10
```

3.5.2 Cluster Management

- Create a cluster with a *deployment*

```
# Create a k8s cluster
maro k8s create ./k8s-azure-create.yml
```

- Scale the cluster

```
  Check `VM Size <https://docs.microsoft.com/en-us/azure/virtual-machines/sizes>`_
↪ to see more node specifications.

# Scale nodes with 'Standard_D4s_v3' specification to 2
maro k8s node scale myK8sCluster Standard_D4s_v3 2

# Scale nodes with 'Standard_D2s_v3' specification to 0
maro k8s node scale myK8sCluster Standard_D2s_v3 0
```

- Delete the cluster

```
# Delete a k8s cluster
maro k8s delete myK8sCluster
```

3.5.3 Run Job

- Push your training image

```
# Push image 'myImage' to the cluster
maro k8s image push myK8sCluster --image-name myImage
```

- Push your training data

```
# Push dqn folder under './myTrainingData/' to a relative path '/myTrainingData'␣
↪in the cluster
# You can then assign your mapping location in the start-job-deployment
maro k8s data push myGrassCluster ./myTrainingData/dqn /myTrainingData
```

- Start a training job with a *deployment*

```
# Start a training job with a start-job-deployment
maro k8s job start myK8sCluster ./k8s-start-job.yml
```

- Or, schedule batch jobs with a *deployment*

```
# Start a training schedule with a start-schedule-deployment
maro k8s schedule start myK8sCluster ./k8s-start-schedule.yml
```

- Get the logs of the job

```
# Logs will be exported to current directory
maro k8s job logs myK8sCluster myJob1
```

- List the current status of the job

```
# List current status of jobs
maro k8s job list myK8sCluster myJob1
```

- Stop a training job

```
# Stop a training job
maro k8s job stop myK8sCluster myJob1
```

3.5.4 Sample Deployments

k8s-aks-create

```
mode: k8s/aks
name: myK8sCluster

cloud:
  subscription: mySubscription
  resource_group: myResourceGroup
  location: eastus
  default_public_key: "{ssh public key}"
  default_username: admin

master:
  node_size: Standard_D2s_v3
```

k8s-start-job

```
mode: k8s/aks
name: myJob1

components:
  actor:
    command: ["python", "{project root}/myTrainingData/dqn/start_actor.py"]
    image: myImage
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 2
      gpu: 0
      memory: 2048M
  learner:
    command: ["python", "{project root}/myTrainingData/dqn/start_learner.py"]
    image: myImage
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048M
```

k8s-start-schedule

```yaml
mode: k8s/aks
name: mySchedule1

job_names:
  - myJob2
  - myJob3
  - myJob4
  - myJob5

components:
  actor:
    command: ["python", "{project root}/myTrainingData/dqn/start_actor.py"]
    image: myImage
    mount:
      target: "{project root}"
    num: 5
    resources:
      cpu: 2
      gpu: 0
      memory: 2048M
  learner:
    command: ["python", "{project root}/myTrainingData/dqn/start_learner.py"]
    image: myImage
    mount:
      target: "{project root}"
    num: 1
    resources:
      cpu: 2
      gpu: 0
      memory: 2048M
```

3.6 Multi-processes Localhost Provisioning

With the following guide, it is easy to implement your training jobs through the multi-processes in the localhost environment.

3.6.1 Prerequisites

- Linux with Python 3.7+
- Redis

3.6.2 Cluster Management

- Get job/schedule deployment template to ./target/path.

```
# Get deployment template
maro process template ./target/path
```

- Create a localhost environment by default parameters.

```
# Create localhost multi-process environment
maro process create
```

- Create a localhost environment with *setting-deployment*.

```
# Get process_setting_deployment to current path.
maro process template --setting_deploy .

maro process create ./process_setting_deployment.yml
```

- Delete a localhost environment.

```
# Delete localhost multi-process environment
maro process delete
```

- Start a training job with *job-deployment*.

```
# Start a training job
maro process job start ./process_job_deployment.yml
```

- Stop a training job.

```
# Stop a training job with job_name
maro process job stop job_name
```

- Delete a job including remove job details in Redis.

```
# Delete a job
maro process job delete job_name
```

- List all jobs.

```
# List all jobs
maro process job list
```

- Get job's log with job_name, dumps to current path.

```
# Get job's log
maro process job logs job_name
```

- Start a schedule with *schedule-deployment*.

```
# Start a schedule
maro process schedule start ./process_schedule_deployment.yml
```

- Stop a schedule.

```
# Stop a schedule with schedule name
maro process schedule stop schedule_name
```

3.6.3 Sample Deployments

process-setting-deployment

```
redis_info:
  host: "localhost"
  port: 19999
redis_mode: MARO      # one of [MARO, customized]. customized Redis won't be exited␣
↪after maro process clear.
parallel_level: 1     # Represented the maximum number of running jobs in the same␣
↪times.
keep_agent_alive: 1   # If 1 represented the agents won't exit until the environment␣
↪delete; otherwise, 0.
agent_countdown: 5    # After agent_countdown times checks, still no jobs will close␣
↪agents. Available only if keep_agent_alive is 0.
check_interval: 60    # The time interval (seconds) of agents check with Redis
```

process-job-deployment

```
mode: process
name: MyJobName

components:
    actor:
        num: 5
        command: "python /target/path/run_actor.py"
    learner:
        num: 1
        command: "python /target/path/run_learner.py"
```

process-schedule-deployment

```
mode: process
name: MyScheduleName

job_names:
    - MyJobName2
    - MyJobName3
    - MyJobName4
    - MyJobName5

components:
    actor:
```

(continues on next page)

(continued from previous page)

```
        num: 5
        command: "python /target/path/run_actor.py"
    learner:
        num: 1
        command: "python /target/path/run_learner.py"
```

3.7 Container Inventory Management (CIM)

The Container Inventory Management (CIM) scenario simulates a common problem of container shipping in marine transportation. Imagine an international market: The goods are packed in containers and shipped by vessels from the exporting country to the importing country. As a result of the imbalanced global trade, the volume of available containers in different ports may not match their needs. In other words, some ports will have excess containers while some ports may be in short. Therefore, We can use the excess capacity on vessels to reposition empty containers to alleviate this imbalance.

3.7.1 Resource Flow

In this scenario, the **container** is the central resource. Two events will trigger the movement of the container:

- The first one is the order, which will lead to the transportation of goods from the source port to the destination port.
- The second one is the repositioning operation. It is used to rebalance the container distribution worldwide.

Order

To simulate a real market, we assume that there will be a certain number of orders from some ports to other ports every day. And the total order number of each day is generated according to a predefined distribution. These orders are then allocated to each export port in a relatively fixed proportion, and each export port will have a relatively fixed number of import ports as customers. The order distribution and the proportion of order allocation are specified in the topology and can be customized based on different requirements.

An order will trigger a life cycle of a container, as shown in the figure above, a life cycle is defined as follows:

- Once an order is generated, an empty container of the corresponding export port (source port) will be released to the shipper.
- The shipper will fill the container with cargo which turns it into a laden and then return it to the port after a few days.
- Loading occurs when the vessel arrives at this port.
- After several days of sailing, the vessel will finally arrive at the corresponding import port (destination port) where the discharging of the laden happens.
- Then the laden will be released to the consignee, and the consignee will take out the cargo in it, which turns it into an empty container again.
- Finally, the consignee returns it as an available container for the import port in a few days.

Container Repositioning

As mentioned above, to rebalance the container distribution, the agent in each port will decide how to reposition the empty containers every time a vessel arrives at the port. The decision consists of two parts:

- Whether to take a `discharge` operation or a `load` operation;
- The number of containers to discharge/load.

The export-oriented ports (e.g. the ports in China) show a clearly high demand feature, and usually require additional supply of empty containers. These ports will tend to discharge empty containers from the vessel if feasible. While the import-oriented ports have a significant surplus feature, that usually receive many empty container from the consignee. So the imported-oriented ports will tend to load the surplus empty containers into the vessel if there is free capacity.

The specific quantity to operate for a `discharge` action is limited by the remaining space in the port and the total number of empty containers in the vessel. Similarly, a `load` action is limited by the remaining space in the vessel and the total number of empty containers in the port. Of course, a good decision will not only consider the self future supply and demand situation, but also the needs and situation of the upstream and downstream ports.

3.7.2 Topologies

To provide an exploration road map from easy to difficult, two kinds of topologies are designed and provided in CIM scenario. Toy topologies provide simplified environments for algorithm debugging and will show some typical relationships between ports to users. We hope these will provide users with some insights to know more and deeper about this scenario. While the global topologies are based on the real-world data, and are bigger and more complicated to present the real problem.

Toy Topologies

(In these topologies, the solid lines indicate the service route (voyage) among ports, while the dashed lines indicate the container flow triggered by orders.)

toy.4p_ssdd_l0.D: There are four ports in this topology. According to the orders, D1 and D2 are simple demanders (the port requiring additional empty containers) while S2 is a simple supplier (the port with surplus empty containers). Although S1 is a simple destination port, it's at the intersection of two service routes, which makes it a special port in this topology. To achieve the global optimum, S1 must learn to distinguish the service routes and take service route specific repositioning operations.

toy.5p_ssddd_l0.D: There are five ports in this topology. According to the orders, D1 and D2 are simple demanders while S1 and S2 are simple suppliers. As a port in the intersection of service routes, although the supply and demand of port T1 can reach a state of self-balancing, it still plays an important role for the global optimum. The best repositioning policy for port T1 is to transfer the surplus empty containers from the left service route to the right one. Also, the port D1 and D2 should learn to discharge only the quantity they need and leave the surplus ones to other ports.

toy.6p_sssbdd_l0.D: There are six ports in this topology. Similar to toy.5p_ssddd, in this topology, there are simple demanders D1 and D2, simple suppliers S1 and S2, and self-balancing ports T1 and T2. More difficult than in toy.5p_ssddd, more transfers should be taken to reposition the surplus empty containers from the left most service route to the right most one, which means a multi-steps solution that involving more ports is required.

Global Topologies

global_trade.22p_l0.D: This is a topology designed based on the real-world data. The order generation model in this topology is built based on the trade data from WTO. According to the query results in WTO from January 2019 to October 2019, The ports with large trade volume are selected, and the proportion of each port as the source of orders is directly proportional to the export volume of the country it belongs to, while the proportion as the destination is proportional to the import volume. In this scenario, there are much more ports, much more service routes. And most ports no longer have a simple supply/demand feature. The cooperation among ports is much more complex and it is difficult to find an efficient repositioning policy manually.

(To make it clearer, the figure above only shows the service routes among ports.)

Naive Baseline

Below are the final environment metrics of the method *no repositioning* and *random repositioning* in different topologies. For each experiment, we setup the environment and test for a duration of 1120 ticks (days).

No Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.4p_ssdd_l0.0	2,240,000	2,190,000	0
toy.4p_ssdd_l0.1	2,240,000	2,190,000	0
toy.4p_ssdd_l0.2	2,240,000	2,190,000	0
toy.4p_ssdd_l0.3	2,239,460	2,189,460	0
toy.4p_ssdd_l0.4	2,244,068	2,194,068	0
toy.4p_ssdd_l0.5	2,244,068	2,194,068	0
toy.4p_ssdd_l0.6	2,244,068	2,194,068	0
toy.4p_ssdd_l0.7	2,244,068	2,194,068	0
toy.4p_ssdd_l0.8	2,241,716	2,191,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.5p_ssddd_l0.0	2,240,000	2,140,000	0
toy.5p_ssddd_l0.1	2,240,000	2,140,000	0
toy.5p_ssddd_l0.2	2,240,000	2,140,000	0
toy.5p_ssddd_l0.3	2,239,460	2,139,460	0
toy.5p_ssddd_l0.4	2,244,068	2,144,068	0
toy.5p_ssddd_l0.5	2,244,068	2,144,068	0
toy.5p_ssddd_l0.6	2,244,068	2,144,068	0
toy.5p_ssddd_l0.7	2,244,068	2,144,068	0
toy.5p_ssddd_l0.8	2,241,716	2,141,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.6p_sssbdd_l0.0	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.1	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.2	2,240,000	2,087,000	0
toy.6p_sssbdd_l0.3	2,239,460	2,086,460	0
toy.6p_sssbdd_l0.4	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.5	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.6	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.7	2,244,068	2,091,068	0
toy.6p_sssbdd_l0.8	2,241,716	2,088,716	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
global_trade.22p_l0.0	2,240,000	1,028,481	0
global_trade.22p_l0.1	2,240,000	1,081,935	0
global_trade.22p_l0.2	2,240,000	1,083,358	0
global_trade.22p_l0.3	2,239,460	1,085,212	0
global_trade.22p_l0.4	2,244,068	1,089,628	0
global_trade.22p_l0.5	2,244,068	1,102,913	0
global_trade.22p_l0.6	2,244,068	1,122,092	0
global_trade.22p_l0.7	2,244,068	1,162,108	0
global_trade.22p_l0.8	2,241,716	1,161,714	0

Random Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.4p_ssdd_l0.0	2,240,000	1,497,138 ± 30,423	4,185,080 ± 185,140
toy.4p_ssdd_l0.1	2,240,000	1,623,710 ± 36,421	2,018,360 ± 36,700
toy.4p_ssdd_l0.2	2,240,000	1,501,466 ± 48,566	2,145,180 ± 90,300
toy.4p_ssdd_l0.3	2,239,460	1,577,011 ± 35,109	2,098,500 ± 35,120
toy.4p_ssdd_l0.4	2,244,068	1,501,835 ± 103,196	2,180,480 ± 33,020
toy.4p_ssdd_l0.5	2,244,068	1,546,227 ± 81,107	2,077,320 ± 113,740
toy.4p_ssdd_l0.6	2,244,068	1,578,863 ± 127,815	2,220,720 ± 106,660
toy.4p_ssdd_l0.7	2,244,068	1,519,495 ± 60,555	2,441,480 ± 79,700
toy.4p_ssdd_l0.8	2,241,716	1,603,063 ± 109,149	2,518,920 ± 193,200

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.5p_ssddd_l0.0	2,240,000	1,371,021 ± 34,619	3,966,120 ± 138,960
toy.5p_ssddd_l0.1	2,240,000	1,720,068 ± 18,939	1,550,280 ± 25,600
toy.5p_ssddd_l0.2	2,240,000	1,716,435 ± 15,499	1,496,860 ± 31,260
toy.5p_ssddd_l0.3	2,239,460	1,700,456 ± 26,510	1,586,640 ± 11,500
toy.5p_ssddd_l0.4	2,244,068	1,663,139 ± 34,244	1,594,160 ± 103,040
toy.5p_ssddd_l0.5	2,244,068	1,681,519 ± 107,863	1,635,360 ± 61,880
toy.5p_ssddd_l0.6	2,244,068	1,660,330 ± 38,318	1,630,060 ± 81,580
toy.5p_ssddd_l0.7	2,244,068	1,709,022 ± 31,440	1,854,340 ± 167,080
toy.5p_ssddd_l0.8	2,241,716	1,763,950 ± 73,935	1,858,420 ± 60,680

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.6p_sssbdd_l0.0	2,240,000	1,529,774 ± 73,104	3,989,560 ± 232,740
toy.6p_sssbdd_l0.1	2,240,000	1,736,385 ± 16,736	1,122,120 ± 28,960
toy.6p_sssbdd_l0.2	2,240,000	1,765,945 ± 4,680	1,052,520 ± 44,020
toy.6p_sssbdd_l0.3	2,239,460	1,811,987 ± 15,436	998,740 ± 69,680
toy.6p_sssbdd_l0.4	2,244,068	1,783,362 ± 39,122	1,059,860 ± 49,100
toy.6p_sssbdd_l0.5	2,244,068	1,755,551 ± 44,855	1,101,100 ± 55,180
toy.6p_sssbdd_l0.6	2,244,068	1,830,504 ± 10,690	1,141,660 ± 10,520
toy.6p_sssbdd_l0.7	2,244,068	1,742,129 ± 23,910	1,311,420 ± 64,560
toy.6p_sssbdd_l0.8	2,241,716	1,761,283 ± 22,338	1,336,540 ± 30,020

Topology	Total Requirement	Resource Shortage	Repositioning Number
global_trade.22p_l0.0	2,240,000	1,010,009 ± 20,942	548,240 ± 14,600
global_trade.22p_l0.1	2,240,000	1,027,395 ± 19,183	188,160 ± 12,940
global_trade.22p_l0.2	2,240,000	1,035,851 ± 4,352	181,240 ± 5,240
global_trade.22p_l0.3	2,239,460	1,032,480 ± 1,332	190,220 ± 8,920
global_trade.22p_l0.4	2,244,068	1,034,412 ± 11,689	186,080 ± 6,280
global_trade.22p_l0.5	2,244,068	1,042,869 ± 16,146	188,720 ± 7,880
global_trade.22p_l0.6	2,244,068	1,096,502 ± 26,896	302,280 ± 27,540
global_trade.22p_l0.7	2,244,068	1,144,981 ± 5,355	283,520 ± 25,700
global_trade.22p_l0.8	2,241,716	1,154,184 ± 7,043	270,960 ± 2,240

3.7.3 Quick Start

Data Preparation

To start a simulation in CIM scenario, no extra data processing is needed. You can just specify the scenario and the topology when initialize an environment and enjoy your exploration in this scenario.

Environment Interface

Before starting interaction with the environment, we need to know the definition of `DecisionEvent` and `Action` in CIM scenario first. Besides, you can query the environment snapshot list to get more detailed information for the decision making.

DecisionEvent

Once the environment need the agent's response to promote the simulation, it will throw an `DecisionEvent`. In the scenario of CIM, the information of each `DecisionEvent` is listed as below:

- **tick** (int): The corresponding tick.
- **port_idx** (int): The id of the port/agent that needs to respond to the environment.
- **vessel_idx** (int): The id of the vessel/operation object of the port/agent.
- **action_scope** (ActionScope): ActionScope has two attributes:
  - `load` indicates the maximum quantity that can be loaded from the port the vessel.
  - `discharge` indicates the maximum quantity that can be discharged from the vessel to the port.
- **early_discharge** (int): When the available capacity in the vessel is not enough to load the ladens, some of the empty containers in the vessel will be early discharged to free the space. The quantity of empty containers that have been early discharged due to the laden loading is recorded in this field.

Action

Once we get a `DecisionEvent` from the environment, we should respond with an `Action`. Valid `Action` could be:

- `None`, which means do nothing.
- A valid `Action` instance, including:
  - **vessel_idx** (int): The id of the vessel/operation object of the port/agent.
  - **port_idx** (int): The id of the port/agent that take this action.
  - **action_type** (ActionType): Whether to load or discharge empty containers in this action.
  - **quantity** (int): The (non-negative) quantity of empty containers to be loaded/discharged.

Example

Here we will show you a simple example of interaction with the environment in random mode, we hope this could help you learn how to use the environment interfaces:

Jump to this notebook for a quick experience.

3.7.4 Visualization

The resource holders in this scenario is the port and vessel. In order to facilitate users to select specific data and observe the overall or partial data trend, the visualization tool provides data selection options in two dimensions: Inter-epoch view & Intra-epoch view.

Inter-epoch view

To change "Start Epoch" and "End Epoch", user could specify the selected data range. To change "Epoch Sampling Ratio", user could change the sampling rate of selected data.

Intra-epoch view

This part shows the data under a selected epoch. By scrolling the slider, users can select different epochs. Furthermore, this part of data is divided into two dimensions: by snapshot and by port according to time and space. In terms of data display, according to the different types of attributes, it is divided into two levels: accumulated data (accumulated attributes. e.g. acc_fulfillment) and detail data.

If user choose to view information by ports, attributes of the selected port would be displayed.

If user choose to view data by snapshots, attributes of selected snapshot would be displayed. The charts and data involved in this part are relatively rich, and we will introduce them by level.

Accumulated Data

This part includes the transfer volume heat map, bar chart of port accumulated attributes and top-k ports of different attributes.

As shown in the following example, the x-axis and y-axis of transfer volume heat map refers to terminal port index and start port index respectively. The rect refers to the volume of cargoes transfer from start port to terminal port. By changing the snapshot index, user could view the dynamic changes in the volume of cargo delivered by the port over time in the current epoch.

The bar chart of Port Accumulated Attributes displays the global change of ports.

Detail Data

Since the cargoes is transported through vessels, information of vessels could be viewed by snapshot. Same as ports, user could change the sampling rate of vessels.

3.8 Bike Repositioning (Citi Bike)

The Citi Bike scenario simulates the bike repositioning problem triggered by the one-way bike trips based on the public trip data from Citi Bike.

> Citi Bike is New York City's bike share system, which consists of a fleet of bikes that are locked into a network of docking stations throughout the city. The bikes can be unlocked from one station and returned to any other station in the system, making them ideal for one-way trips. People use bike share to commute to work or school, run errands, get to appointments or social engagements, and more.

Since the demand for bikes and empty docks is dynamically changed during a day, and the bike flow between two stations are not equal in a same period, some stations suffer from severe bike shortages, while some have too much bikes and too few empty docks. In such a situation, the bike repositioning is essential to balance the bike's supply and demand. A good bike repositioning can not only meet the needs in the stations with heavy ride demand but also free the stations that do not have enough empty docks. Also, in the long run, a good bike repositioning can improve the bike usability, empower citizens' daily life, and reduce the carbon emission.

3.8.1 Resource Flow

In this scenario, the **bike** is the central resource. Two events will trigger the movement of the bike:

- The first one is the trip requirement, which may cause the bike transfer from the source station to the destination station;
- The second one is the repositioning operation. It is used to rebalance the bike distribution among stations.

Bike Trip

In the citi bike scenario in MARO, the trip generation and the corresponding bike flow is defined as follows:

- Given a fixed time interval, for each specific source-destination station pair, a trip requirement will arise according to a predefined distribution or the real trip data. It depends on the chosen topology.
- If there are enough available bikes in the source station of the trip requirement, the required bike(s) will be unlocked and assigned to this trip. Otherwise, a shortage will be recorded in the source station.
- The trip duration is read from the trip log if real trip data is used. Otherwise, the duration will be sampled from a specific random distribution.
- At the end of the trip, the bike will be returned to the destination station. But if the destination does not have enough available docks, the bike will be returned to the nearest station with available docks.

Bike Repositioning

As for the repositioning operation, the simulator in MARO will regularly check the remaining bikes in each station and compare it with a predefined low watermark and high watermark. If the bike inventory is lower than the low watermark, the station will generate a `Demand` event to request the supply of bikes from its neighboring stations. Similarly, if the bike inventory is higher than the high watermark, the station will generate a `Supply` event to transport excess bikes to its neighboring stations. The low watermark and the high watermark is specified in the topology and can be customized based on different requirements.

The target station candidates of the `Supply` and `Demand` events are selected by a predefined multi-layer filter in this scenario:

1. The distance between the caller station and the neighboring stations will be used to filter and get a specific number of stations;
2. The number of available bikes at each candidate station will be used to further filter on the candidate stations. For a `Supply` event, the stations with less bikes will be kept, while for a `Demand` event, the stations with more bikes will be kept;
3. The future trip requirement of the target station will be the last filter. For a `Supply` event, the stations with more future trip requirement will be left in the final station candidate set, while the stations with less future trip requirement will be left for `Demand` event.

The size of the candidate sets in each filter level is specified in the topology and can be customized based on different requirements.

Once the target station candidate is filtered, the `action scope` for each candidate will also be calculated in the simulator and return to the decision agent together with some other information in the *DecisionEvent*. For a `Supply` event, the bike inventory of the caller station and the number of available docks of the target station candidates will be attached. On the contrary, for a `Demand` event, the number of available docks of the source station and the bike inventory of the target station candidates will be attached.

Based on the given target station candidates and the corresponding `action scope`, the decision agent of the caller station should decide how many bikes to transfer to/request from the target station. We call a pair of `(target station, bike number)` a repositioning action. After an action taken, the destination station should wait for a certain period to get the bikes available for trip requirement. The action lead time is sampled from a predefined distribution.

3.8.2 Topologies

To provide an exploration road map from easy to difficult, two kinds of topologies are designed and provided in Citi Bike scenario. Toy topologies provide a super simplified environment for algorithm debugging, while the real topologies with real data from Citi Bike historical trips can present the real problem to users.

Toy Topologies

In toy topology, the generation of the trip requirements follows a stable pattern as introduced above. The detailed trip demand pattern are listed as below. And we hope that these toy topologies can provide you with some insights about this scenario.

toy.3s_4t: There are three stations in this topology. Every two minutes, there will be a trip requirement from S2 to S3 and a trip requirement from S3 to S2. At the same time, every two minutes, the system will generate trip requirement from S1 to S3 and from S1 to S2 with a fixed probability (80% and 20%, respectively). In this topology, the traffic flow between S2 and S3 is always equal, but station S1 is a super consumer with only bikes flow out. So the best repositioning policy in this topology is to reposition bikes from S2 and S3 to S1. It requires the active request action of S1 or the proactive transfer action of S2 and S3.

toy.4s_4t: There are four stations in this topology. According to the global trip demand, there are more returned bikes than leaving bikes in station S1 and S3, while more leaving bikes than returned bikes in station S2 and S4. So the best repositioning policy in this topology is to reposition the excess bikes from S1 and S3 to S2 and S4. Furthermore, the cooperation between these stations is also necessary since only a proper allocation can lead to a globally optimal solution.

toy.5s_6t: There are five stations in this topology. Although trip demand is more complex than the other two topologies above, we can still find that station S1 is a self-balancing station, station S2 and S5 have more returned bikes, and station S3 and S4 have more leaving bikes. Just like in topology toy.4s_4t, the best repositioning policy is to reposition excess bikes from S2 and S5 to S3 and S4 coordinately.

Real Topologies

ny.YYYYMM: Different from the stable generation model in the toy topologies, the trip requirement in the topology ny.YYYYMM is generated based on the real trip data from Citi Bike, which includes the source station, the destination station, and the duration of each trip. Besides, the total number of available bikes in this kind of topologies is counted from the real trip data of the specific month. Weighted by the the latest capacity of each stations, the available bikes are allocated to each station, which constitutes the initial bike inventory of each station. In this series of topologies, the definition of the bike flow and the trigger mechanism of repositioning actions are the same as those in the toy topologies. We provide this series of topologies to better simulate the actual Citi Bike scenario.

Naive Baseline

Below are the final environment metrics of the method *no repositioning* and *random repositioning* in different topologies. For each experiment, we setup the environment and test for a duration of 1 week.

No Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.3s_4t	15,118	8,233	0
toy.4s_4t	9,976	7,048	0
toy.5s_6t	16,341	9,231	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201801	48,089	2,688	0
ny.201802	126,374	8,814	0
ny.201803	138,952	10,942	0
ny.201804	161,443	10,349	0
ny.201805	323,375	29,081	0
ny.201806	305,971	26,412	0
ny.201807	254,715	19,669	0
ny.201808	302,589	26,352	0
ny.201809	313,002	28,472	0
ny.201810	339,268	24,109	0
ny.201811	263,227	21,485	0
ny.201812	209,102	15,876	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201901	161,474	10,775	0
ny.201902	187,354	12,593	0
ny.201903	148,371	7,193	0
ny.201904	280,852	16,906	0
ny.201905	287,290	27,213	0
ny.201906	379,415	33,968	0
ny.201907	309,365	21,105	0
ny.201908	371,969	33,703	0
ny.201909	344,847	24,528	0
ny.201910	351,855	29,544	0
ny.201911	324,327	29,489	0
ny.201912	184,015	14,205	0

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.202001	169,304	12,449	0
ny.202002	206,105	14,794	0
ny.202003	235,986	15,436	0
ny.202004	91,810	2,348	0
ny.202005	169,412	5,231	0
ny.202006	197,883	7,608	0

Random Repositioning

Topology	Total Requirement	Resource Shortage	Repositioning Number
toy.3s_4t	15,154	8,422 ± 11	449 ± 22
toy.4s_4t	10,186	4,371 ± 72	3,392 ± 83
toy.5s_6t	16,171	7,513 ± 40	3,242 ± 71

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201801	48,089	6,693 ± 138	445,996 ± 6,756
ny.201802	126,374	21,418 ± 120	446,564 ± 3,505
ny.201803	138,952	22,121 ± 272	448,259 ± 1,831
ny.201804	161,443	22,201 ± 194	453,705 ± 3,697
ny.201805	323,375	54,365 ± 538	470,771 ± 5,337
ny.201806	305,971	49,876 ± 1,091	481,443 ± 6,981
ny.201807	254,715	46,199 ± 204	483,788 ± 982
ny.201808	302,589	53,679 ± 433	485,137 ± 2,557
ny.201809	313,002	61,432 ± 75	474,851 ± 2,908
ny.201810	339,268	64,269 ± 600	461,928 ± 1,018
ny.201811	263,227	40,440 ± 239	467,050 ± 6,595
ny.201812	209,102	26,067 ± 234	457,173 ± 6,444

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.201901	161,474	19,295 ± 155	444,445 ± 2,287
ny.201902	187,354	23,875 ± 282	456,888 ± 362
ny.201903	148,371	12,451 ± 312	409,226 ± 5,392
ny.201904	280,852	29,591 ± 170	464,671 ± 6,148
ny.201905	287,290	44,199 ± 542	485,077 ± 6,140
ny.201906	379,415	51,396 ± 256	503,503 ± 4,742
ny.201907	309,365	33,861 ± 643	500,443 ± 4,314
ny.201908	371,969	51,319 ± 417	516,684 ± 1,400
ny.201909	344,847	34,532 ± 466	476,965 ± 3,932
ny.201910	351,855	37,828 ± 502	496,135 ± 4,167
ny.201911	324,327	34,745 ± 427	484,599 ± 8,771
ny.201912	184,015	20,119 ± 110	437,311 ± 5,936

Topology	Total Requirement	Resource Shortage	Repositioning Number
ny.202001	169,304	17,152 ± 218	476,821 ± 1,052
ny.202002	206,105	24,223 ± 209	480,012 ± 1,547
ny.202003	235,986	23,749 ± 654	458,536 ± 1,457
ny.202004	91,810	3,349 ± 48	326,817 ± 3.131
ny.202005	169,412	10,177 ± 216	378,038 ± 2,429
ny.202006	197,883	11,741 ± 170	349,932 ± 4,375

3.8.3 Quick Start

Data Preparation

To start the simulation of Citi Bike scenario, users have two options for the data preparation:

- If the topology data is not generated in advance, the system will automatically download and process the relevant data when the environment is created. The data will be stored in a temporary folder and be automatically deleted after the experiment.
- Before creating the environment, users can also manually download and generate relevant data. This approach will save you a lot of time if you need to conduct several experiments on the same topology. Therefore, we encourage you to generate the relevant data manually first.

The following is the introduction to related commands:

Environment List Command

The data environment `list` command is used to list the environments that need the data files generated before the simulation.

```
maro env data list

scenario: citi_bike, topology: ny.201801
scenario: citi_bike, topology: ny.201802
scenario: citi_bike, topology: ny.201803
scenario: citi_bike, topology: ny.201804
scenario: citi_bike, topology: ny.201805
scenario: citi_bike, topology: ny.201806
...
```

Generate Command

The data `generate` command is used to automatically download and build the specified predefined scenario and topology data files for the simulation. Currently, there are three arguments for the data `generate` command:

- `-s`: required, used to specify the predefined scenario. Valid scenarios are listed in the result of *environment list command*.
- `-t`: required, used to specify the predefined topology. Valid topologies are listed in the result of *environment list command*.
- `-f`: optional, if set, to force to re-download and re-generate the data files and overwrite the already existing ones.

```
maro env data generate -s citi_bike -t ny.201802

The data files for citi_bike-ny201802 will then be downloaded and deployed to ~/.maro/
↪data/citibike/_build/ny201802.
```

For the example above, the directory structure should be like:

```
|-- ~/.maro
    |-- data
    |   |-- citi_bike
    |       |-- .build          # bin data file
    |           |-- [topology]  # topology
    |       |-- .source
    |           |-- .download   # original data file
    |           |-- .clean      # cleaned data file
    |-- temp                    # download temp file
```

Build Command

The data `build` command is used to build the CSV data files to binary data files that the simulator needs. Currently, there are three arguments for the data `build` command:

- `--meta`: required, used to specify the path of the meta file. The source columns that to be converted and the data type of each columns should be specified in the meta file.
- `--file`: required, used to specify the path of the source CSV data file(s). If multiple source CSV data files are needed, you can list all the full paths of the source files in a specific file and use a `@` symbol to specify it.
- `--output`: required, used to specify the path of the target binary file.

```
maro data build --meta ~/.maro/data/citibike/meta/trips.yml --file ~/.maro/data/
↪citibike/source/_clean/ny201801/trip.csv --output ~/.maro/data/citibike/_build/
↪ny201801/trip.bin
```

Environment Interface

Before starting interaction with the environment, we need to know the definition of `DecisionEvent` and `Action` in Citi Bike scenario first. Besides, you can query the environment snapshot list to get more detailed information for the decision making.

DecisionEvent

Once the environment need the agent's response to reposition bikes, it will throw an `DecisionEvent`. In the scenario of Citi Bike, the information of each `DecisionEvent` is listed as below:

- **station_idx** (int): The id of the station/agent that needs to respond to the environment.
- **tick** (int): The corresponding tick.
- **frame_index** (int): The corresponding frame index, that is the index of the corresponding snapshot in the environment snapshot list.
- **type** (DecisionType): The decision type of this decision event. In Citi Bike scenario, there are 2 types:
  - `Supply` indicates there is too many bikes in the corresponding station, so it is better to reposition some of them to other stations.

  - Demand indicates there is no enough bikes in the corresponding station, so it is better to reposition bikes from other stations.

- **action_scope** (dict): A dictionary that maintains the information for calculating the valid action scope:
  - The key of these item indicate the station/agent ids.
  - The meaning of the value differs for different decision type:
    - If the decision type is Supply, the value of the station itself means its bike inventory at that moment, while the value of other target stations means the number of their empty docks.
    - If the decision type is Demand, the value of the station itself means the number of its empty docks, while the value of other target stations means their bike inventory.

Action

Once we get a DecisionEvent from the environment, we should respond with an Action. Valid Action could be:

- None, which means do nothing.
- A valid Action instance, including:
  - **from_station_idx** (int): The id of the source station of the bike transportation.
  - **to_station_idx** (int): The id of the destination station of the bike transportation.
  - **number** (int): The quantity of the bike transportation.

Example

Here we will show you a simple example of interaction with the environment in random mode, we hope this could help you learn how to use the environment interfaces:

```
from maro.simulator import Env
from maro.simulator.scenarios.citi_bike.common import Action, DecisionEvent,␣
↪DecisionType

import random

# Initialize an environment of Citi Bike scenario, with a specific topology.
# In CitiBike, 1 tick means 1 minute, durations=1440 here indicates a length of 1 day.
# In CitiBike, one snapshot will be maintained every snapshot_resolution ticks,
# snapshot_resolution=30 here indicates 1 snapshot per 30 minutes.
env = Env(scenario="citi_bike", topology="toy.3s_4t", start_tick=0, durations=1440,␣
↪snapshot_resolution=30)

# Query for the environment summary, the business instances and intra-instance␣
↪attributes
# will be listed in the output for your reference.
print(env.summary)

metrics: object = None
decision_event: DecisionEvent = None
is_done: bool = False
action: Action = None

num_episode = 2
```

(continues on next page)

(continued from previous page)

```
for ep in range(num_episode):
    # Gym-like step function.
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_2hour_frames = [
            x for x in range(decision_event.frame_index - 4, decision_event.frame_
↪index)
        ]
        decision_station_idx = decision_event.station_idx
        intr_station_infos = ["trip_requirement", "bikes", "shortage"]

        # Query the snapshot list of this environment to get the information of
        # the trip requirements, bikes, shortage of the decision station in the past␣
↪2 hours.
        past_2hour_info = env.snapshot_list["stations"][
            past_2hour_frames : decision_station_idx : intr_station_infos
        ]

        if decision_event.type == DecisionType.Supply:
            # Supply: the value of the station itself means the bike inventory.
            self_bike_inventory = decision_event.action_scope[decision_event.station_
↪idx]
            # Supply: the value of other stations means the quantity of empty docks.
            target_idx_dock_tuple_list = [
                (k, v) for k, v in decision_event.action_scope.items() if k !=␣
↪decision_event.station_idx
            ]
            # Randomly choose a target station weighted by the quantity of empty␣
↪docks.
            target_idx, target_dock = random.choices(
                target_idx_dock_tuple_list,
                weights=[item[1] for item in target_idx_dock_tuple_list],
                k=1
            )[0]
            # Generate the corresponding random Action.
            action = Action(
                from_station_idx=decision_event.station_idx,
                to_station_idx=target_idx,
                number=random.randint(0, min(self_bike_inventory, target_dock))
            )

        elif decision_event.type == DecisionType.Demand:
            # Demand: the value of the station itself means the quantity of empty␣
↪docks.
            self_available_dock = decision_event.action_scope[decision_event.station_
↪idx]
            # Demand: the value of other stations means their bike inventory.
            target_idx_inventory_tuple_list = [
                (k, v) for k, v in decision_event.action_scope.items() if k !=␣
↪decision_event.station_idx
            ]
            # Randomly choose a target station weighted by the bike inventory.
            target_idx, target_inventory = random.choices(
                target_idx_inventory_tuple_list,
                weights=[item[1] for item in target_idx_inventory_tuple_list],
                k=1
```

(continues on next page)

(continued from previous page)

```
            )[0]
            # Generate the corresponding random Action.
            action = Action(
                from_station_idx=target_idx,
                to_station_idx=decision_event.station_idx,
                number=random.randint(0, min(self_available_dock, target_inventory))
            )

        else:
            action = None

        # Drive the environment with the random action.
        metrics, decision_event, is_done = env.step(action)

    # Query for the environment business metrics at the end of each episode,
    # it is usually users' optimized object (usually includes multi-target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

Jump to this notebook for a quick experience.

3.8.4 Visualization

The resource holder in this scenario is the bike station. The number of epoches is varied based on selected strategy. Currently, we generate result with Greedy Policy. Only one epoch is included in result. Thus, inter-epoch view would not be displayed.

Inter-epoch view

This part display cross-epoch summary information. User could select the Start Epoch and End Epoch, as well as Epoch Sampling Ratio. Since there is only one epoch, this part is hidden.

Intra-epoch view

Intra-view is divided into two dimensions according to time and space.

If user choose to view information by station, it means that attributes of all snapshots within a selected station would be displayed. By changing the option “station index”, user could view data of different stations. By changing the option “Snapshot Sampling Ratio”, Users can freely adjust the sampling rate. For example, if there are 100 snapshots and user selected 0.3 as sampling ratio, 30 snapshots data would be selected to render the chart.

To be specific, the line chart could be customized with operations in the following example.

By choosing the item “All”, all of attributes would be displayed. In addition, according to the data characteristics of each scenario, users will be provided with the option to quickly select a set of data.

e.g. In this scenario, item “Requirement Info” refers to [trip_requirement, shortage, fulfillment].

Moreover, to improve the flexibility of visualizing data, user could use pre-defined formula and selected attributes to generate new attributes. Generated attributes would be treated in the same way as original attributes.

If user choose to view information by snapshot, it means attributes of all stations within a selected snapshot would be displayed. By changing option “snapshot index”, user could view data of different snapshot. By changing option “Station Sampling Ratio”, user could change the number of sampled data.

Particularly, if user want to check the name of a specific station, just hovering on the according bar.

3.9 Virtual Machine Scheduling (VM Scheduling)

In 21th century, the business needs of cloud computing are dramatically increasing. During the cloud service, users request Virtual Machine (VM) with a certain amount of resources (eg. CPU, memory, etc). The following important issue is how to allocate the physical resources for these VMs? The VM Scheduling scenario aims to find a better solution of the VM scheduling problem in cloud data centers. Now, consider a specific time, the number of VM requests and arrival pattern is fixed. Given a cluster of limited physical machines(PM) with limited physical resources, different VM allocation strategies result in different amount of successful completion and different operating cost of the data center. For cloud providers, a good VM allocation strategy can maximize the resource utilization and thus can increase the profit by providing more VMs to users. For cloud users, a good VM allocation strategy can minimize the VM response time and have a better using experience. We hope this scenario can meet the real needs and provide you with a demand simulation that is closest to the real situation.

3.9.1 Resource Flow

In this scenario, the physical resources in each PM are the central resource, which currently includes the physical cores and memory. A full resource life cycle always contains the steps below:

- Coming VM requests ask for a certain amount of resources. Resource requirements are varied based on the different VM requests.
- Based on the scheduling agent’s strategy, the VM will be allocated to and be created in a specified PM as long as that PM’s remaining resources are enough.
- The VM’s resource utilization changes dynamically and the PM’s real-time energy consumption will be simulated in the runtime simulation.
- After a period of execution, the VM completes its tasks. The simulator will release the resources allocated to this VM and deallocate this VM from the PM. Finally, the resource is free and is ready to serve the next VM request.

VM Request

In the VM scheduling scenario, the VM requests are uniformly sampled from real workloads. As long as the original dataset is large enough and the sample ratio is not too small, the sampled VM requests can follow a similar distribution to the original ones.

Given a fixed time interval, a VM request will arise according to the real VM workload data. The request contains the VM information, such as the subscription id, the deployment id, and the VM category, VM’s required resources, including the required CPU cores and the required memory, and the remaining buffer time.

- Whenever receive a VM request, the MARO simulator will first calculate the remaining resources of each PM, filtering out the valid PMs (valid PMs means that the remaining resources of PM are enough for the required resources of the VM).
- Then, the simulator delivers all valid PMs and the required resources of the awaiting VM to the VM scheduler (agent) for a decision.

We have two categories of VM. One is interactive, and the other one is delay-insensitive.

- Interactive: The interactive VMs usually require low response time, so we set this kind of VMs can only be allocated to the non-oversubscribable PM server.
- Delay-insensitive: The delay-insensitive VMs usually serve for batch-tasks or development workload. This kind of VMs can be allocated to the over-subscribable PM server.

VM Allocation

Based on the valid PM list, the historical information recorded by the simulator, and the detailed required resources of the VM, the VM scheduler (decision agent) will make the decision according to its allocation strategy.

There are two types of meaningful actions:

- Deliver a valid PM ID to the simulator.
- Postpone the VM request, which will leave this request to be handled later if the remaining buffer time is enough.

See the detailed attributes of *Action*.

Oversubscription

To maximize each PM's utilization, cloud providers will oversubscribe the physical resource. Considering the various service level, the physical machines are then divided into the over-subscribable ones and non-oversubscribable ones. For the over-subscription, there are several parameters can be set in the config.yml. In our scenario, there are two resources could be oversubscribed, CPU and memory, so we have two maximum oversubscription rate can be set.

- `MAX_CPU_OVERSUBSCRIPTION_RATE`: The oversubscription rate of CPU. For example, the default setting is 1.15, that means each PM can be allocated at most 1.15 times of its resource capacity.
- `MAX_MEM_OVERSUBSCRIPTION_RATE`: The oversubscription rate of memory. Similar to the CPU rate.

To protect the PM from the overloading, we need to consider the CPU utilization. The `MAX_UTILIZATION_RATE` is used as the security mechanism, that can be set in the config.yml.

- `MAX_UTILIZATION_RATE`: The default setting is 1, which means that when filtering the valid PMs, the maximum allowed physical CPU utilization is 100%.

Runtime Simulation

Dynamic Utilization

To make the simulated environment closest to the real situation. We also simulate the resource utilization (currently only CPU utilization) of each VM. The CPU utilization of the VM varies every tick based on the real VM workload readings. We will also regularly update the real-time resource utilization of each PM based on the live VMs in it.

Real-time Energy Consumption

One of the most important characteristics that cloud providers concern is the energy consumption of the data center. The different VM allocation can result in different energy consumption of the PM cluster, we also simulate the energy usage based on the CPU utilization.

Energy Curve

As we mention before, the lower energy consumption of the PMs, the lower cost to maintain the physical servers. In our simulation, we currently use a non-linear energy curve like the one in the above figure to simulate the energy based on the CPU utilization.

Overload

Since the VM's CPU utilization varies along the time, when enabling the oversubscription, it might happen that the sum of VM's CPU usage exceed the capacity of the physical resource. This situation called overload.

Overloading may lead to VM's performance degradation or service level agreements (SLAs) violations in real production (We will support these features in the future). Currently, for the situation of overloading, we only support quiescing (killing) all VMs or just recording the times of overloading, which can also be set in config.yml.

- `KILL_ALL_VMS_IF_OVERLOAD`: If this action is enable, once overloading happens, all VMs located at the overloading PMs will be deallocated. To consider the effect of overloading, we will still count the energy consumed by the high utilization. The impact of the quiescing action on the PM's utilization will be reflected in the next tick.

No matter enable killing all VMs or not, we will calculate the number of overload PMs and the number of overload VMs. These two metrics are cumulative values and will be recorded as the environment metrics.

VM Deallocation

The MARO simulator regularly checks the finished VMs in every tick. A finished VM means that it goes through a complete life cycle, is ready to be terminated, and the resources it occupies will be available again in the end. The simulator will then release the finished VM's resources, and finally remove the VM from the PM.

3.9.2 Topologies

Azure Topologies

The original data comes from Azure public dataset. The dataset contains real Azure VM workloads, including the information of VMs and their utilization readings in 2019 lasting for 30 days. Total number of VM recorded is 2,695,548.

In our scenario, we pre-processed the AzurePublicDatasetV2. The detailed information of the data schema can be found here. After pre-processed, the data contains

- Renumbered VM ID
- VM cores and memory(GB) requirements
- Real VM creation and deletion time (converted to the tick, 1 tick means 5 minutes in real time)

As for the utilization readings part, we sort the renumbered VM ID and CPU utilization pairs by the timestamp (tick).

To provide system workloads from light to heavy, two kinds of simple topologies are designed and provided in VM Scheduling scenario.

azure.2019.10k

Uniformly random sample.

- Total number of VMs: 10,000
- Average number of concurrent VMs: 835.7
- Average number of CPU cores requested: 3.8
- Average memory requested: 15.9 GB
- Average CPU utilization: 15.7 %

PM setting (Given by the /[topologies]/config.yml):

- Amount: 100
- CPU Cores: 32
- Memory: 128 GB

azure.2019.336k

Uniformly random sample.

- Total number of VMs: 336,000
- Average number of concurrent VMs: 28,305.9
- Average number of CPU cores requested: 3.8
- Average memory requested: 16.1 GB
- Average CPU utilization: 15.6 %

PM setting (Given by the /[topologies]/config.yml):

- Amount: 880
- CPU Cores: 16
- Memory: 112 GB

Naive Baseline

Belows are the final environment metrics of the method **Random Allocation** and **Best-Fit Allocation** in different topologies. For each experiment, we setup the environment and test for a duration of 30 days. Besides, we use several settings of PM capacity to test performance under different initial resources.

Random Allocation

Randomly allocate to a valid PM.

Topology	PM Setting	Total VM Requests	Total Energy Consumption	Successful Allocation	Successful completion	Failed Allocation
Azure.2019.10k	100 PMs, 32 Cores, 128 GB	10,000	2,430,651.6	9,850	9,030	150
	100 PMs, 16 Cores, 112 GB	10,000	2,978,445.0	8,011	7,411	1,989
Azure.2019.336k	880 PMs, 32 Cores, 128 GB	335,985	26,681,249.7	176,468	165,715	159,517
	880 PMs, 16 Cores, 112 GB	335,985	26,367,238.7	92,885	87,153	243,100

Best-Fit Allocation

Choose the valid PM with the least remaining resources (only consider CPU cores here).

Topology	PM Setting	Total VM Requests	Total Energy Consumption	Successful Allocation	Successful completion	Failed Allocation
Azure.2019.10k	100 PMs, 32 Cores, 128 GB	10,000	2,395,328.7	10,000	9,180	0
	100 PMs, 16 Cores, 112 GB	10,000	2,987,086.6	7,917	7,313	2,083
Azure.2019.336k	880 PMs, 32 Cores, 128 GB	335,985	26,695,470.8	171,044	160,495	164,941
	880 PMs, 16 Cores, 112 GB	335,985	26,390,972.9	92,263	86,600	243,722

3.9.3 Quick Start

Data Preparation

When the environment is first created, the system will automatically trigger the pipeline to download and process the data files. Afterwards, if you want to run multiple simulations, the system will detect whether the processed data files exist or not. If not, it will then trigger the pipeline again. Otherwise, the system will reuse the processed data files.

Customize Dataset

If you want to use your own dataset, you need to prepare two csv files, `vm_table` and `cpu_readings_file`. Below is the data schema and the format that you need to follow.

- `vm_table`:
  - vm_id: int. The id number of VMs.
  - sub_id: int. The subscription id of VMs.
  - deploy_id: int. The deployment id of VMs.
  - timestamp: int. The timestamp of VM's creation time.

  - vm_lifetime: int. The lifetime of VMs. Lifetime equals the deletion time - creation time (timestamp) + 1.
  - vm_deleted: int. The timestamp of VM's deletion time.
  - vm_category: int. The category of VMs. Currently, we have three categories of VM:
    * `Delay-Insensitive`: The VMs workload that could be delayed, such as batch tasks or test workload. This kind of VMs could be allocated to the over-subscribable PM. Store as `0`.
    * `Interactive`: The VMs workload that are interactive, which need user response in time. This kind of VMs could only allocated to the non-oversubscribable PMs. Store as `1`.
    * `Unknown`. Unknown types. To avoid the overloading, this kind of VMs are treated as the interactive ones, which could only allocated to the non-oversubscribable PMs. Store as `2`.
  - vm_cpu_cores: int. The CPU cores of VMs.
  - vm_memory: int. The memory of VMs.
- `cpu_readings_file`:
  - timestamp: int. The timestamp. Should be matched with the timestamp in `vm_table`.
  - vm_id: int. The id number of VMs. Should be matched with the ids in `vm_table`.
  - cpu_utilization: float. The utilization of VM CPU. Store in the unit of percentage (%).

Build Command

After preparing two files as the above formats. You need to convert them to binary files. We provide the `build` command to build your own CSV dataset to binary files that the MARO simulator can use. Currently, there are three required arguments for the data `build` command:

- `--meta`: required, used to specify the path of the meta file. In default, the meta files are under

  ```
  ~/.maro/data/vm_scheduling/meta/
  ```

  The source columns that to be converted and the data type of each columns should be specified in the meta file.
- `--file`: required, used to specify the path of the source CSV data file(s). If multiple source CSV data files are needed, you can list all the full paths of the source files in a specific file and use a `@` symbol to specify it.
- `--output`: required, used to specify the path of the target binary file.

```
maro data build --meta $PATH_TO_META_FILE --file $PATH_TO_CSV_FILE  --output $PATH_TO_
↪OUTPUT_FILE
```

For example,

```
maro data build --meta ~/.maro/data/vm_scheduling/meta/vmtable.yml  --file ~/.maro/
↪data/vm_scheduling/.build/azure.2019.10k/vmtable.bin --output $PWD/vmtable.bin
```

After building the binary files. Specify the direct path of `VM_TABLE` and `CPU_READINGS` in the config.yml under the topologies directories. Then you can use your own dataset to run the simulation.

Environment Interface

Before starting interaction with the environment, we need to know the definition of `DecisionPayload` and `Action` in VM Scheduling scenario first. Besides, you can query the environment snapshot list to get more detailed information for the decision making.

DecisionPayload

Once the environment need the agent's response to promote the simulation, it will throw an `PendingDecision` event with the `DecisionPayload`. In the scenario of VM Scheduling, the information of `DecisionPayload` is listed as below:

- **valid_pms** (List[int]): The list of the PM ID that is considered as valid (Its CPU and memory resource is enough for the incoming VM request).
- **vm_id** (int): The VM ID of the incoming VM request (VM request that is waiting for the allocation).
- **vm_cpu_cores_requirement** (int): The CPU cores that is requested by the incoming VM request.
- **vm_memory_requirement** (int): The memory resource that is requested by the incoming VM request.
- **remaining_buffer_time** (int): The remaining buffer time for the VM allocation. The VM request will be treated as failed when the remaining_buffer_time is spent. The initial buffer time budget can be set in the config.yml.

Action

Once get a `PendingDecision` event from the environment, the agent should respond with an Action. Valid `Action` includes:

- **None**. It means do nothing but ignore this VM request.
- `AllocateAction`: If the MARO simulator receives the `AllocateAction`, the VM's creation time will be fixed at the tick it receives. Besides, the simulator will update the workloads (the workloads include CPU cores, the memory, and the energy consumption) of the target PM. The `AllocateAction` includes:
  - vm_id (int): The ID of the VM that is waiting for the allocation.
  - pm_id (int): The ID of the PM where the VM is scheduled to allocate to.
- `PostponeAction`: If the MARO simulator receives the `PostponeAction`, it will calculate the remaining buffer time.
  - If the time is still enough, the simulator will re-generate a new request event and insert it to the corresponding tick (based on the `Postpone Step` and `DELAY_DURATION`). The `DecisionPayload` of the new requirement event only differs in the remaining buffer time from the old ones.
  - If the time is exhausted, the simulator will note it as a failed allocation.

  The `PostponeAction` includes:
  - vm_id (int): The ID of the VM that is waiting for the allocation.
  - postpone_step (int): The number of times that the allocation to be postponed. The unit is `DELAY_DURATION`. 1 means delay 1 `DELAY_DURATION`, which can be set in the config.yml.

Example

Here we will show you a simple example of interaction with the environment in random mode, we hope this could help you learn how to use the environment interfaces:

```python
import random

from maro.simulator import Env
from maro.simulator.scenarios.vm_scheduling import AllocateAction, DecisionPayload, 
↪PostponeAction

# Initialize an Env for vm_scheduling scenario
env = Env(
  scenario="vm_scheduling",
  topology="azure.2019.10k",
  start_tick=0,
  durations=8638,
  snapshot_resolution=1
)

metrics: object = None
decision_event: DecisionPayload = None
is_done: bool = False
action: AllocateAction = None

# Start the env with a None Action
metrics, decision_event, is_done = env.step(None)

while not is_done:
    valid_pm_num: int = len(decision_event.valid_pms)
    if valid_pm_num <= 0:
        # No valid PM now, postpone.
        action: PostponeAction = PostponeAction(
            vm_id=decision_event.vm_id,
            postpone_step=1
        )
    else:
        # Randomly choose an available PM.
        random_idx = random.randint(0, valid_pm_num - 1)
        pm_id = decision_event.valid_pms[random_idx]
        action: AllocateAction = AllocateAction(
            vm_id=decision_event.vm_id,
            pm_id=pm_id
        )
    metrics, decision_event, is_done = env.step(action)

print(f"[Random] Topology: azure.2019.10k. Total ticks: 8638. Start tick: 0")
print(metrics)
```

Jump to this notebook for a quick experience.

3.10 Command support for scenarios

After installation, MARO provides a command that generate project for user, make it much easier to use or customize scenario.

```
maro project new
```

This command will show a step-by-step wizard to create a new project under current folder. Currently it supports 2 modes.

3.10.1 1. Use built-in scenarios

To use built-in scenarios, please agree the first option "Use built-in scenario" with "yes" or "y", default is "yes". Then you can select a built-in scenario and topologies with auto-completing.

```
Use built-in scenario?yes
Scenario name:cim
Use built-in topology (configuration)?yes
Topology name to use:global_trade.22p_l0.0
Durations to emulate:1024
Number of episodes to emulate:500
{'durations': 1024,
'scenario': 'cim',
'topology': 'global_trade.22p_l0.0',
'total_episodes': 500,
'use_builtin_scenario': True,
'use_builtin_topology': True}

Is this OK?yes
```

If these settings correct, then this command will create a runner.py script, you can just run with:

```
python runner.py
```

This script contains minimal code to interactive with environment without any action, you can then extend it as you wish.

Also you can create you own topology (configuration) if you say "no" for options "Use built-in topology (configuration)?". It will ask you for a name of new topology, then copy the content from built-in one into your working folder (topologies/your_topology_name/config.yml).

3.10.2 2. Customized scenario

This mode is used to generate a template of customize scenario for you instead of writing it from scratch. To enable this, say "no" for option "Use built-in scenario", then provide your scenario name, default is current folder name.

```
Use built-in scenario?no
New scenario name:my_test
New topology name:my_test
Durations to emulate:1000
Number of episodes to emulate:100
{'durations': 1000,
'scenario': 'my_test',
'topology': 'my_test',
```

(continues on next page)

(continued from previous page)

```
'total_episodes': 100,
'use_builtin_scenario': False,
'use_builtin_topology': False}

Is this OK?yes
```

This will generate following files like below:

```
-- runner.py
-- scenario
    -- business_engine.py
    -- common.py
    -- events.py
    -- frame_builder.py
    -- topologies
        -- my_test
            -- config.yml
```

The script "runner.py" is the entry of this project, it will interactive with your scenario without action. Then you can fill "scenario/business_engine.py" with your own logic.

3.11 Greedy Policy for Citi Bike

In this example we demonstrate using a simple greedy policy for Citi Bike, which is a real-world bike repositioning scenario.

3.11.1 Greedy Policy

Our greedy policy is simple: if the event type is supply, the policy will make the current station send as many bikes as possible to one of k stations with the most empty docks. If the event type is demand, the policy will make the current station request as many bikes as possible from one of k stations with the most bikes. We use a heap data structure to find the top k supply/demand candidates from the action scope associated with each decision event.

```
class GreedyPolicy:
    ...
    def choose_action(self, decision_event: DecisionEvent):
        if decision_event.type == DecisionType.Supply:
            """
            Find k target stations with the most empty slots, randomly choose one of␣
↪them and send as many bikes to
            it as allowed by the action scope
            """
            top_k_demands = []
            for demand_candidate, available_docks in decision_event.action_scope.
↪items():
                if demand_candidate == decision_event.station_idx:
                    continue

                heapq.heappush(top_k_demands, (available_docks, demand_candidate))
                if len(top_k_demands) > self._demand_top_k:
                    heapq.heappop(top_k_demands)

            max_reposition, target_station_idx = random.choice(top_k_demands)
```

(continues on next page)

(continued from previous page)

```
            action = Action(decision_event.station_idx, target_station_idx, max_
↪reposition)
        else:
            """
            Find k source stations with the most bikes, randomly choose one of them␣
↪and request as many bikes from
            it as allowed by the action scope.
            """
            top_k_supplies = []
            for supply_candidate, available_bikes in decision_event.action_scope.
↪items():
                if supply_candidate == decision_event.station_idx:
                    continue

                heapq.heappush(top_k_supplies, (available_bikes, supply_candidate))
                if len(top_k_supplies) > self._supply_top_k:
                    heapq.heappop(top_k_supplies)

            max_reposition, source_idx = random.choice(top_k_supplies)
            action = Action(source_idx, decision_event.station_idx, max_reposition)

        return action
```

3.11.2 Interaction with the Greedy Policy

This environment is driven by real trip history data from Citi Bike.

```
env = Env(scenario=config.env.scenario, topology=config.env.topology, start_
↪tick=config.env.start_tick,
          durations=config.env.durations, snapshot_resolution=config.env.resolution)

if config.env.seed is not None:
    env.set_seed(config.env.seed)

policy = GreedyPolicy(config.agent.supply_top_k, config.agent.demand_top_k)
metrics, decision_event, done = env.step(None)
while not done:
    metrics, decision_event, done = env.step(policy.choose_action(decision_event))

env.reset()
```

Note: All related code snippets are supported in maro playground.

3.12 Simulation Toolkit

MARO simulation toolkit provides some predefined environments, such as CIM, Citi Bike, etc. It also provides some critical and reusable wheels for easily building new scenarios, with a high-performance guarantee and uniformed programming paradigm, including Data Model, Event Buffer, and Business Engine.

3.12.1 Overview

The key features of MARO simulation toolkit:

- Event-driven, which is friendly to business logging data, and more in line with real-world scenarios;
- High execution performance;
- Low development cost for new environments;
- Convenient interface for spatial-temporal data accessing, which is friendly to decision-related information querying.

In order to achieve the key features listed above, we choose Python as the frontend language. In the meantime, to avoid Python GIL problem and to achieve the better performance of memory allocation and cache hitting, we choose Cython as the backend language. Furthermore, we provide a declarative Data Model to auto-generate the related data class wrapper for easy underlying memory access, a Snapshot List slicing interface to quickly accessing data from the spatial-temporal and intra-node perspective. Considering the variously complex scenarios, we decide to build up an event-oriented simulation system, which is not only suitable to feed existing business logging data but also convenient to inject random events. New scenarios can be implemented in the pluggable Business Engine based on the Event Buffer supporting. The following figure illustrates the overall architecture of the MARO simulation system.

3.12.2 Environment

Well-designed topologies are provided in each predefined scenario (i.e. CIM, Citi Bike). You can specify both the scenario and the topology when initializing an environment. To distinguish the complicated problems in the real world, we provide various topologies with different difficulties for the user to do a step-by-step exploration. In general, the interface of environments is Gym-like, which use `step` to trigger the environment to apply `actions` from `agent(s)`. Furthermore, we concern more about the decision mode of multi-agent/hierarchical-agent and how to conveniently query data on the levels of spatial, temporal, and intra-node (resource holder). The following shows a typical environment interaction workflow and code snippet.

```python
from maro.simulator import Env
from maro.simulator.scenarios.cim.common import Action

# Initialize an environment with a specific scenario, related topology.
# In Container Inventory Management, 1 tick means 1 day, here durations=100 means a␣
↪length of 100 days
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0", start_tick=0, durations=100)

# Query environment summary, which includes business instances, intra-instance␣
↪attributes, etc.
print(env.summary)

for ep in range(2):
    # Gym-like step function.
```

(continues on next page)

(continued from previous page)

```
    metrics, decision_event, is_done = env.step(None)

    while not is_done:
        past_week_ticks = [
            x for x in range(decision_event.tick - 7, decision_event.tick)
        ]
        decision_port_idx = decision_event.port_idx
        intr_port_infos = ["booking", "empty", "shortage"]

        # Query the snapshot list of the environment to get the information of
        # the booking, empty container inventory, shortage of the decision port in␣
↪the past week.
        past_week_info = env.snapshot_list["ports"][
            past_week_ticks : decision_port_idx : intr_port_infos
        ]

        dummy_action = Action(
            vessel_idx=decision_event.vessel_idx,
            port_idx=decision_event.port_idx,
            quantity=0
        )

        # Drive environment with dummy action (no repositioning).
        metrics, decision_event, is_done = env.step(dummy_action)

    # Query environment business metrics at the end of an episode,
    # it is your optimized object (usually includes multi-target).
    print(f"ep: {ep}, environment metrics: {env.metrics}")
    env.reset()
```

3.12.3 Critical Components

These components are the foundation of the whole MARO simulation system and can be easily reused to build up various real-world business scenarios with good performance and uniformed pattern. You can find more details in Data Model, Event Buffer, and Business Engine.

3.13 Data Model

The data model of MARO provides a declarative interface. We choose Python as the frontend language for saving development cost, and we take C as the backend language for improving the execution reference. What's more, the backend store is a pluggable design, user can choose different backend implementation based on their real performance requirement and device limitation.

Currently there are two data model backend implementation: static and dynamic. Static implementation used Numpy as its data store, do not support dynamic attribute length, the advance of this version is that its memory size is same as its declaration. Dynamic implementation is hand-craft c++. It supports dynamic attribute (list) which will take more memory than the static implementation but is faster for querying snapshot states and accessing attributes.

3.13.1 Key Concepts

As shown in the figure above, there are some key concepts in the data model:

- **Node** is the abstraction of the resource holder, which is usually the major business instance of the scenario (i.e. vessels and ports in CIM scenario). A node usually has various attributes to present the business nature.
- **(Slot) Attribute** is the abstraction of business properties for the scenarios-specific resource holder (node). The attributes of a node could be declared as different data types based on the real requirements. Furthermore, for each attribute, a `slot` feature is provided to support the fixed-size array. The `slot` number can indicate the attribute values (e.g. the three different container types in CIM scenario) or the detailed categories (e.g. the ten specific products in the *Use Case* below). By default, the `slot` value is one. As for the dynamic backend implementation, an attribute can be marked as is_list or is_const to identify it is a list attribute or a const attribute respectively. A list attribute's default slot number is 0, and can be increased as demand, max number is 2^32. A const attribute is designed for the value that will not change after initialization, e.g. the capacity of a port/station. The value is shared between frames and will not be copied when taking a snapshot.
- **Frame** is the collection of all nodes in the environment. The historical frames present the aggregated state of the environment during a specific period, while the current frame hosts the latest state of the environment at the current time point.
- **Snapshot List** is the dumped frames based on a pre-defined resolution. It captures the aggregated changes of the environment between the dump points.

3.13.2 Use Case

- Below is the declaration of a retail frame, which includes warehouse and store nodes.

```
from maro.backends.backend import AttributeType
from maro.backends.frame import node, NodeAttribute, NodeBase, FrameNode,␣
↪FrameBase

TOTAL_PRODUCT_CATEGORIES = 10
TOTAL_STORES = 8
TOTAL_WAREHOUSES = 2
TOTAL_SNAPSHOT = 100


@node("warehouse")
class Warehouse(NodeBase):
    inventories = NodeAttribute(AttributeType.Int, TOTAL_PRODUCT_CATEGORIES)
    shortages = NodeAttribute(AttributeType.Int, TOTAL_PRODUCT_CATEGORIES)

    def __init__(self):
        self._init_inventories = [100 * (i + 1) for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
        self._init_shortages = [0] * TOTAL_PRODUCT_CATEGORIES

    def reset(self):
        self.inventories[:] = self._init_inventories
        self.shortages[:] = self._init_shortages


@node("store")
class Store(NodeBase):
```

(continues on next page)

(continued from previous page)

```
    inventories = NodeAttribute(AttributeType.Int, TOTAL_PRODUCT_CATEGORIES)
    shortages = NodeAttribute(AttributeType.Int, TOTAL_PRODUCT_CATEGORIES)
    sales = NodeAttribute(AttributeType.Int, TOTAL_PRODUCT_CATEGORIES)

    def __init__(self):
        self._init_inventories = [10 * (i + 1) for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
        self._init_shortages = [0] * TOTAL_PRODUCT_CATEGORIES
        self._init_sales = [0] * TOTAL_PRODUCT_CATEGORIES

    def reset(self):
        self.inventories[:] = self._init_inventories
        self.shortages[:] = self._init_shortages
        self.sales[:] = self._init_sales


class RetailFrame(FrameBase):
    warehouses = FrameNode(Warehouse, TOTAL_WAREHOUSES)
    stores = FrameNode(Store, TOTAL_STORES)

    def __init__(self):
        # If your actual frame number was more than the total snapshot number,␣
↪the old snapshots would be rolling replaced.
        # You can select a backend implementation that will fit your requirement.
        super().__init__(enable_snapshot=True, total_snapshot=TOTAL_SNAPSHOT,␣
↪backend_name="static/dynamic")
```

- The operations on the retail frame.

```
retail_frame = RetailFrame()

# Fulfill the initialization values to the backend memory.
for store in retail_frame.stores:
    store.reset()

# Fulfill the initialization values to the backend memory.
for warehouse in retail_frame.warehouses:
    warehouse.reset()

# Take a snapshot of the first tick frame.
retail_frame.take_snapshot(0)
snapshot_list = retail_frame.snapshots
print(f"Max snapshot list capacity: {len(snapshot_list)}")

# Query sales, inventory information of all stores at first tick, len(snapshot_
↪list["store"]) equals to TOTAL_STORES.
all_stores_info = snapshot_list["store"][0::["sales", "inventories"]].
↪reshape(TOTAL_STORES, -1)
print(f"All stores information at first tick (numpy array): {all_stores_info}")

# Query shortage information of first store at first tick.
first_store_shortage = snapshot_list["store"][0:0:"shortages"]
print(f"First store shortages at first tick (numpy array): {first_store_shortage}
↪")

# Query inventory information of all warehouses at first tick, len(snapshot_list[
↪"warehouse"]) equals to TOTAL_WAREHOUSES.
```

(continues on next page)

(continued from previous page)

```
all_warehouses_info = snapshot_list["warehouse"][0::"inventories"].reshape(TOTAL_
↪WAREHOUSES, -1)
print(f"All warehouses information at first tick (numpy array): {all_warehouses_
↪info}")

# Add fake shortages to first store.
retail_frame.stores[0].shortages[:] = [i + 1 for i in range(TOTAL_PRODUCT_
↪CATEGORIES)]
retail_frame.take_snapshot(1)

# Query shortage information of first and second store at first and second tick.
store_shortage_history = snapshot_list["store"][[0, 1]: [0, 1]: "shortages"].
↪reshape(2, -1)
print(f"First and second store shortage history at the first and second tick␣
↪(numpy array): {store_shortage_history}")
```

3.13.3 Supported Attribute Data Type

All supported data types for the attribute of the node:

Attribute Data Type	C Type	Range
Attribute.Byte	char	-128 .. 127
Attribute.UByte	unsigned char	0 .. 255
Attribute.Short (i2)	short	-32,768 .. 32,767
Attribute.UShort	unsigned short	0 .. 65,535
Attribute.Int (i4)	int32_t	-2,147,483,648 .. 2,147,483,647
Attribute.UInt (i4)	uint32_t	0 .. 4,294,967,295
Attribute.Long (i8)	int64_t	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
Attribute.ULong (i8)	uint64_t	0 .. 18,446,744,073,709,551,615
Attribute.Float (f)	float	-3.4E38 .. 3.4E38
Attribute.Double (d)	double	-1.7E308 .. 1.7E308

3.13.4 Advanced Features

For better data access, we also provide some advanced features, including:

- **Attribute value change handler**: It is a hook function for the value change event on a specific attribute. The member function with the `_on_{attribute_name}_changed` naming pattern will be automatically invoked when the related attribute value changed. Below is the example code:

```
from maro.backends.frame import node, NodeBase, NodeAttribute

@node("test_node")
class TestNode(NodeBase):
    test_attribute = NodeAttribute("i")

    def _on_test_attribute_changed(self, value: int):
        pass
```

- **Snapshot list slicing**: It provides a slicing interface for querying temporal (frame), spatial (node), intra-node (attribute) information. Both a single index and an index list are supported for querying specific frame(s), node(s), and attribute(s), while the empty means querying all. The return value is a flattened 1-dimension NumPy array, which aligns with the slicing order as below:

```
snapshot_list = env.snapshot_list

# Get max size of snapshots (in memory).
print(f"Max snapshot size: {len(snapshot_list)}")

# Get snapshots of a specific node type.
test_nodes_snapshots = snapshot_list["test_nodes"]

# Get node instance amount.
print(f"Number of test_nodes in the frame: {len(test_nodes_snapshots)}")

# Query one attribute on all frames and nodes.
states = test_nodes_snapshots[::"int_attribute"]

# Query two attributes on all frames and nodes.
states = test_nodes_snapshots[::["int_attribute", "float_attribute"]]

# Query one attribute on all frame and the first node.
states = test_nodes_snapshots[:0:"int_attribute"]

# Query attribute by node index list.
states = test_nodes_snapshots[:[0, 1, 2]:"int_attribute"]

# Query one attribute on the first frame and the first node.
states = test_nodes_snapshots[0:0:"int_attribute"]

# Query attribute by frame index list.
states = test_nodes_snapshots[[0, 1, 2]: 0: "int_attribute"]

# The querying states is different between static and dynamic implementation
# Static implementation will return a 1-dim numpy array, as the shape is known␣
↪according to the parameters.
# Dynamic implementation will return a 4-dim numpy array, that shape is (ticks,␣
↪node_indices, attributes, slots).
# Usually we can just flatten the state from dynamic implementation, then it will␣
↪be same as static implementation,
# except for list attributes.
# List attribute only support one tick, one node index and one attribute name to␣
↪query, cannot mix with normal attributes
states = test_nodes_snapshots[0: 0: "list_attribute"]

# Also with dynamic implementation, we can get the const attributes which is␣
↪shared between snapshot list, even without
# any snapshot (need to provided one tick for padding).
states = test_nodes_snapshots[0: [0, 1]: ["const_attribute", "const_attribute_2"]]
```

3.13.5 States in built-in scenarios' snapshot list

Currently there are 3 ways to expose states in built-in scenarios:

Summary

Summary(env.summary) is used to expose static states to outside, it provide 3 items by default: node_mapping, node_detail and event payload.

The "node_mapping" item usually contains node name and related index, but the structure may be different for different scenario.

The "node_detail" usually used to expose node definitions, like node name, attribute name and slot number, this is useful if you want to know what attributes are support for a scenario.

The "event_payload" used show that payload attributes of event in scenario, like "RETURN_FULL" event in CIM scenario, it contains "src_port_idx", "dest_port_idx" and "quantity".

Metrics

Metrics(env.metrics) is designed that used to expose raw states of reward since we have removed reward support in v0.2 version, and it also can be used to export states that not supported by snapshot list, like dictionary or complex structures. Currently there are 2 ways to get the metrics from environment: env.metrics, or 1st result from env.step.

This metrics usually is a dictionary with several keys, but this is determined by business engine.

Snapshot_list

Snapshot list is the history of nodes (or data model) for a scenario, it only support numberic data types now. It supported slicing query with a numpy array, so it support batch operations, make it much faster than using raw python objects.

Nodes and attributes may different for different scenarios, following we will introduce about those in built-in scenarios.

NOTE: Per tick state means that the attribute value will be reset to 0 after each step.

3.13.6 CIM

Default settings for snapshot list

Snapshot resolution: 1

Max snapshot number: same as durations

Nodes and attributes in scenario

In CIM scenario, there are 3 node types:

port

capacity

type: int slots: 1

The capacity of port for stocking containers.

empty

type: int slots: 1

Empty container volume on the port.

full

type: int slots: 1

Laden container volume on the port.

on_shipper

type: int slots: 1

Empty containers, which are released to the shipper.

on_consignee

type: int slots: 1

Laden containers, which are delivered to the consignee.

shortage

type: int slots: 1

Per tick state. Shortage of empty container at current tick.

acc_storage

type: int slots: 1

Accumulated shortage number to the current tick.

booking

type: int slots: 1

Per tick state. Order booking number of a port at the current tick.

acc_booking

type: int slots: 1

Accumulated order booking number of a port to the current tick.

fulfillment

type: int slots: 1

Fulfilled order number of a port at the current tick.

acc_fulfillment

type: int slots: 1

Accumulated fulfilled order number of a port to the current tick.

transfer_cost

type: float slots: 1

Cost of transferring container, which also covers loading and discharging cost.

vessel

capacity

type: int slots: 1

The capacity of vessel for transferring containers.

NOTE: This attribute is ignored in current implementation.

empty

type: int slots: 1

Empty container volume on the vessel.

full

type: int slots: 1

Laden container volume on the vessel.

remaining_space

type: int slots: 1

Remaining space of the vessel.

early_discharge

type: int slots: 1

Discharged empty container number for loading laden containers.

route_idx

type: int slots: 1

Which route current vessel belongs to.

last_loc_idx

type: int slots: 1

Last stop port index in route, it is used to identify where is current vessel.

next_loc_idx

type: int slots: 1

Next stop port index in route, it is used to identify where is current vessel.

past_stop_list

type: int slots: dynamic

NOTE: This and following attribute are special, that its slot number is determined by configuration, but different with a list attribute, its slot number is fixed at runtime.

Stop indices that we have stopped in the past.

past_stop_tick_list

type: int slots: dynamic

Ticks that we stopped at the port in the past.

future_stop_list

type: int slots: dynamic

Stop indices that we will stop in the future.

future_stop_tick_list

type: int slots: dynamic

Ticks that we will stop in the future.

matrices

Matrices node is used to store big matrix for ports, vessels and containers.

full_on_ports

type: int slots: port number * port number

Distribution of full from port to port.

full_on_vessels

type: int slots: vessel number * port number

Distribution of full from vessel to port.

vessel_plans

type: int slots: vessel number * port number

Planed route info for vessels.

How to

How to use the matrix(s)

Matrix is special that it only have one instance (index 0), and the value is saved as a flat 1 dim array, we can reshape it after querying.

```
# assuming that we want to use full_on_ports attribute.

tick = 0

# we can get the instance number of a node by calling the len method
port_number = len(env.snapshot_list["port"])

# this is a 1 dim numpy array
full_on_ports = env.snapshot_list["matrices"][tick::"full_on_ports"]

# reshape it, then this is a 2 dim array that from port to port.
full_on_ports = full_on_ports.reshape(port_number, port_number)
```

3.13.7 Citi-Bike

Default settings for snapshot list

Snapshot resolution: 60

Max snapshot number: same as durations

Nodes and attributes in scenario

station

bikes

type: int slots: 1

How many bikes avaiable in current station.

shortage

type: int slots: 1

Per tick state. Lack number of bikes in current station.

trip_requirement

type: int slots: 1

Per tick states. How many requirements in current station.

fulfillment

type: int slots: 1

How many requirement is fit in current station.

capacity

type: int slots: 1

Max number of bikes this station can take.

id

type: int slots: 1

Id of current station.

weekday

type: short slots: 1

Weekday at current tick.

temperature

type: short slots: 1

Temperature at current tick.

weather

type: short slots: 1

Weather at current tick.

0: sunny, 1: rainy, 2: snowy 3: sleet.

holiday

type: short slots: 1

If it is holidy at current tick.

0: holiday, 1: not holiday

extra_cost

type: int slots: 1

Cost after we reach the capacity after executing action, we have to move extra bikes to other stations.

transfer_cost

type: int slots: 1

Cost to execute action to transfer bikes to other station.

failed_return

type: int slots: 1

Per tick state. How many bikes failed to return to current station.

min_bikes

type: int slots: 1

Min bikes number in a frame.

matrices

trips_adj

type: int slots: station number * station number

Used to store trip requirement number between 2 stations.

3.13.8 VM-scheduling

Default settings for snapshot list

Snapshot resolution: 1

Max snapshot number: same as durations

Nodes and attributes in scenario

Cluster

id

type: short slots: 1

Id of the cluster.

region_id

type: short slots: 1

Region is of current cluster.

data_center_id

type: short slots: 1

Data center id of current cluster.

total_machine_num

type: int slots: 1

Total number of machines in the cluster.

empty_machine_num

type: int slots: 1

The number of empty machines in this cluster. A empty machine means that its allocated CPU cores are 0.

data_centers

id

type: short slots: 1

Id of current data center.

region_id

type: short slots: 1

Region id of current data center.

zone_id

type: short slots: 1

Zone id of current data center.

total_machine_num

type: int slots: 1

Total number of machine in current data center.

empty_machine_num

type: int slots: 1

The number of empty machines in current data center.

pms

Physical machine node.

id

type: int slots: 1

Id of current machine.

cpu_cores_capacity

type: short slots: 1

Max number of cpu core can be used for current machine.

memory_capacity

type: short slots: 1

Max number of memory can be used for current machine.

pm_type

type: short slots: 1

Type of current machine.

cpu_cores_allocated

type: short slots: 1

How many cpu core is allocated.

memory_allocated

type: short slots: 1

How many memory is allocated.

cpu_utilization

type: float slots: 1

CPU utilization of current machine.

energy_consumption

type: float slots: 1

Energy consumption of current machine.

oversubscribable

type: short slots: 1

Physical machine type: non-oversubscribable is -1, empty: 0, oversubscribable is 1.

region_id

type: short slots: 1

Region id of current machine.

zone_id

type: short slots: 1

Zone id of current machine.

data_center_id

type: short slots: 1

Data center id of current machine.

cluster_id

type: short slots: 1

Cluster id of current machine.

rack_id

type: short slots: 1

Rack id of current machine.

Rack

id

type: int slots: 1

Id of current rack.

region_id

type: short slots: 1

Region id of current rack.

zone_id

type: short slots: 1

Zone id of current rack.

data_center_id

type: short slots: 1

Data center id of current rack.

cluster_id

type: short slots: 1

Cluster id of current rack.

total_machine_num

type: int slots: 1

Total number of machines on this rack.

empty_machine_num

type: int slots: 1

Number of machines that not in use on this rack.

regions

id

type: short slots: 1

Id of curent region.

total_machine_num

type: int slots: 1

Total number of machines in this region.

empty_machine_num

type: int slots: 1

Number of machines that not in use in this region.

zones

id

type: short slots: 1

Id of this zone.

total_machine_num

type: int slots: 1

Total number of machines in this zone.

empty_machine_num

type: int slots: 1

Number of machines that not in use in this zone.

3.14 Event Buffer

Event buffer is the execution engine of the MARO simulation system. Various business scenarios are constructed by different business event series and the related handler functions. To implement a scenario-specific environment, users need to provide the definitions of business events and related handler functions. After that, the input events of the event buffer will be automatically processed based on their priority. The priority of an event is first determined by its declared execution tick. Events of the same tick will be executed according to the FIFO rule. Currently, only a single-thread version event buffer is provided.

```
# Initialize an event buffer.
event_buffer: EventBuffer = EventBuffer()

# Insert a mock event.
event_buffer.insert_event(event)

# Execute events at a specific tick.
executed_events = event_buffer.execute(tick)
```

3.14.1 Event Category

To simplify the implementation of the business logic, MARO provides two kinds of basic event types, which can be used to construct various event execution pattern:

- **Atom event** is an event without any dependence. An atom event will be immediately popped out from the event buffer after execution.
- **Cascade event** is a series of events with dependencies. An internal event queue is hosted for the sub-events of each cascade event. Only after all sub-events executed, a cascade event can be popped out from the event buffer. Furthermore, the cascade event is chainable, which means that events in the internal event queue can also be cascade events.

3.14.2 Event Format

We provide a general-purpose event format for all the scenarios in MARO. A legal event generally contains the following properties:

- **tick** (int): The execution tick of this event.
- **event_type** (int): The type of this event. It is a customized field, the default value is 0 (PREDEFINE_EVENT_ACTION).
- **source** (str): The id/code of the event generator (not implemented yet).
- **target** (str): The id/code of the event receiver (not implemented yet).
- **payload** (Object): It is used to transfer sufficient information to the event handler.
- **tag** (EventCategory): The tag of the *event category*. Valid values include ATOM and CASCADE.
- **immediate_event_list** (list): The sub-event queue of the cascade event. Atom event does not have this field.
- **state** (EventState): The state of this event. Valid values include PENDING, EXECUTING and FINISHED. The figure below indicates the state changing of an event:

3.15 Business Engine

MARO provides a pluggable mechanism to isolate the business logic and the simulation kernel parts. For different business scenarios, specific business engines should be bind to it.

The business engine is responsible for defining:

- **Business instance**. Generally, the business instances are the resource holders in the business logic. For example:
  - The ports and vessels in the CIM scenario;
  - The stations in the Citi Bike scenario;
- **Business event**. Since it is closely related to the actual business logic, the business events are reproduced based on the existing business log data or generated according to the predefined business rules. For example:
  - The receiving of customers' orders, the vessel departure, etc in the CIM scenario;
  - The receiving of the customers' trip requirements, the finishing of a trip, etc in the Citi Bike scenario;
- **Decision event**. It is the external interactive interface for the resource optimization. The environment returns the decision events to the decision agent to trigger the agent's actions. For example:

  - The empty container repositioning operations that triggered by a vessel arrival event in the CIM scenario;
  - The bike repositioning operations that triggered when there are too many or too few bikes remained in the station in Citi Bike scenario;
- **Optimize metrics**. There are often multiple goals in real business scenarios, these metrics should be defined so as to be recorded to evaluate the repositioning. For example:
  - The shortage that used to measure the number of un-satisfied requirements;
  - The cost of the repositioning operations;

As shown in the figure below, the simulator will load the scenario-specific business engine and convert the filled business time series data into the corresponding business events. After that, the entire simulation system would be driven by these business events. In short, with the uniformed business engine interface, the simulation of different business scenarios is only based on the pluggable business engine (scenario-specific).

Generally, the business time series data is read from the historical log or generated by a data generation model. Currently, for topologies in Citi Bike scenario, data processing is needed before starting the simulation. You can find the brief introduction of the data processing command in Data Processing.

3.16 RL Toolkit

MARO provides a full-stack abstraction for reinforcement learning (RL) which includes various customizable components. In order to provide a gentle introduction for the RL toolkit, we cover the components in a top-down manner, starting from the learning workflow.

3.16.1 Workflow

The nice thing about MARO's RL workflows is that it is abstracted neatly from business logic, policies and learning algorithms, making it applicable to practically any scenario that utilizes standard reinforcement learning paradigms. The workflow is controlled by a main process that executes 2-phase learning cycles: roll-out and training (Fig. 3.1). The roll-out phase collects data from one or more environment simulators for training. There can be a single environment simulator located in the same thread as the main loop, or multiple environment simulators running in parallel on a set of remote workers (Fig. 3.2) if you need to collect large amounts of data fast. The training phase uses the data collected during the roll-out phase to train models involved in RL policies and algorithms. In the case of multiple large models, this phase can be made faster by having the computationally intensive gradient-related tasks sent to a set of remote workers for parallel processing (Fig. 3.3).

Fig. 3.1: Learning Workflow

Fig. 3.2: Parallel Roll-out

Fig. 3.3: Distributed Training

3.16.2 Environment Sampler

An environment sampler is an entity that contains an environment simulator and a set of policies used by agents to interact with the environment (Fig. 3.4). When creating an RL formulation for a scenario, it is necessary to define an environment sampler class that includes these key elements:

- how observations / snapshots of the environment are encoded into state vectors as input to the policy models. This is sometimes referred to as state shaping in applied reinforcement learning;
- how model outputs are converted to action objects defined by the environment simulator;
- how rewards / penalties are evaluated. This is sometimes referred to as reward shaping.

In parallel roll-out, each roll-out worker should have its own environment sampler instance.

Fig. 3.4: Environment Sampler

3.16.3 Policy

`Policy` is the most important concept in reinforcement learning. In MARO, the highest level abstraction of a policy object is `AbsPolicy`. It defines the interface `get_actions()` which takes a batch of states as input and returns corresponding actions. The action is defined by the policy itself. It could be a scalar or a vector or any other types. Env sampler should take responsibility for parsing the action to the acceptable format before passing it to the environment.

The simplest type of policy is `RuleBasedPolicy` which generates actions by pre-defined rules. `RuleBasedPolicy` is mostly used in naive scenarios. However, in most cases where we need to train the policy by interacting with the environment, we need to use `RLPolicy`. In MARO's design, a policy cannot train itself. Instead, polices could only be trained by *Algorithm (Trainer)* (we will introduce trainer later in this page). Therefore, in addition to `get_actions()`, `RLPolicy` also has a set of training-related interfaces, such as `step()`, `get_gradients()` and `set_gradients()`. These interfaces will be called by trainers for training. As you may have noticed, currently we assume policies are built upon deep learning models, so the training-related interfaces are specifically designed for gradient descent.

`RLPolicy` is further divided into three types: - `ValueBasedPolicy`: For valued-based policies. - `DiscretePolicyGradient`: For gradient-based policies that generate discrete actions. - `ContinuousPolicyGradient`: For gradient-based policies that generate continuous actions.

The above classes are all concrete classes. Users do not need to implement any new classes, but can directly create a policy object by configuring parameters. Here is a simple example:

```
ValueBasedPolicy(
    name="policy",
    q_net=MyQNet(state_dim=128, action_num=64),
)
```

For now, you may have no idea about the `q_net` parameter, but don't worry, we will introduce it in the next section.

3.16.4 Model

The above code snippet creates a `ValueBasedPolicy` object. Let's pay attention to the parameter `q_net`. `q_net` accepts a `DiscreteQNet` object, and it serves as the core part of a `ValueBasedPolicy` object. In other words, `q_net` defines the model structure of the Q-network in the value-based policy, and further determines the policy's behavior. `DiscreteQNet` is an abstract class, and `MyQNet` is a user-defined implementation of `DiscreteQNet`. It can be a simple MLP, a multi-head transformer, or any other structure that the user wants.

MARO provides a set of abstractions of basic & commonly used PyTorch models like `DiscereteQNet`, which enables users to implement their own deep learning models in a handy way. They are:

- `DiscreteQNet`: For `ValueBasedPolicy`.
- `DiscretePolicyNet`: For `DiscretePolicyGradient`.
- `ContinuousPolicyNet`: For `ContinuousPolicyGradient`.

Users should choose the proper types of models according to the type of policies, and then implement their own models by inheriting the abstract ones (just like `MyQNet`).

There are also some other models for training purposes. For example:

- `VNet`: Used in the critic part in the actor-critic algorithm.
- `MultiQNet`: Used in the critic part in the MADDPG algorithm.
- ...

The way to use these models is exactly the same as the way to use the policy models.

3.16.5 Algorithm (Trainer)

When introducing policies, we mentioned that policies cannot train themselves. Instead, they have to be trained by external algorithms, which are also called trainers. In MARO, a trainer represents an RL algorithm, such as DQN, actor-critic, and so on. These two concepts are equivalent in the MARO context. Trainers take interaction experiences and store them in the internal memory, and then use the experiences in the memory to train the policies. Like `RLPolicy`, trainers are also concrete classes, which means they could be used by configuring parameters. Currently, we have 4 trainers (algorithms) in MARO:

- `DiscreteActorCriticTrainer`: Actor-critic algorithm for policies that generate discrete actions.
- `DiscretePPOTrainer`: PPO algorithm for policies that generate discrete actions.
- `DDPGTrainer`: DDPG algorithm for policies that generate continuous actions.
- `DQNTrainer`: DQN algorithm for policies that generate discrete actions.
- `DiscreteMADDPGTrainer`: MADDPG algorithm for policies that generate discrete actions.

Each trainer has a corresponding `Param` class to manage all related parameters. For example, `DiscreteActorCriticParams` contains all parameters used in `DiscreteActorCriticTrainer`:

```
@dataclass
class DiscreteActorCriticParams(TrainerParams):
    get_v_critic_net_func: Callable[[], VNet] = None
    reward_discount: float = 0.9
    grad_iters: int = 1
    critic_loss_cls: Callable = None
    clip_ratio: float = None
    lam: float = 0.9
    min_logp: Optional[float] = None
```

An example of creating an actor-critic trainer:

```
DiscreteActorCriticTrainer(
    name='ac',
    params=DiscreteActorCriticParams(
        get_v_critic_net_func=lambda: MyCriticNet(state_dim=128),
        reward_discount=.0,
        grad_iters=10,
        critic_loss_cls=torch.nn.SmoothL1Loss,
        min_logp=None,
        lam=.0
    )
)
```

In order to indicate which trainer each policy is trained by, in MARO, we require that the name of the policy start with the name of the trainer responsible for training it. For example, policy `ac_1.policy_1` is trained by the trainer named `ac_1`. Violating this provision will make MARO unable to correctly establish the corresponding relationship between policy and trainer.

More details and examples can be found in the code base (link).

As a summary, the relationship among policy, model, and trainer is demonstrated in Fig. 3.5:

Fig. 3.5: Summary of policy, model, and trainer

3.17 Distributed Toolkit

MARO distributed toolkit provides a unified, fast, and infrastructure-independent interface to support RL distributed training.

As shown in the overall architecture diagram above, MARO distributed toolkit follows a message-passing pattern that the cooperation between different components is based on the messages sending and receiving. A typical master/worker distributed program usually contains the following steps:

1. The master component will send tasks(w/ or w/o data) to the worker components;
2. The worker components will finish the tasks in their local computing environments or the local devices;
3. The worker components return the computed results to the master component.

According to the actual needs, the communication mode between master and worker components can be synchronous or asynchronous.

3.17.1 Key Components

There are two key components in the distributed toolkit:

- **Communication**: It provides the general message passing interfaces, such as `(i)send`, `receive`, `(i)broadcast`, `(i)scatter`, etc. The communication component use a replaceable communication protocol driver to adopt different communication protocol stack (e.g. TCP/IP, InfiniBand). Check the distributed communication to get more details.

- **Orchestration**: It primarily provides a unified interface for cluster management and job management on different infrastructures. Check the distributed orchestration to get more details.

3.18 Distributed Communication

The distributed communication component provides a general message passing mechanism, which is used to build various communication topologies in the distributed environment. Besides the essential communication primitive supporting, it also provides the functions of peer discovering, fault recovering (partially), conditional event auto-dispatching, etc.

3.18.1 Proxy

Providing an implementation of the communication primitives, proxy is the primary entity of the communication component. Proxy provides a uniformed communication interface, the underlying driver is pluggable based on the real requirements. Currently, we use ZeroMQ as the default choice. Proxy also provides support for peer discovering based on Redis.

Message

Message is designed for general purpose, it is used to package the communication content between components. The main attributes of a message instance include:

- `tag`: A customized attribute, it can be used to implement the auto-dispatching logic with a *conditional event register table*.
- `source`: The alias of the message sender.
- `destination`: The alias of the message receiver.
- `payload`: A Python object for remote function call.
- `session_id` (auto-generated): UUID of a specific session, one session may include multiple messages.
- `message_id` (auto-generated): UUID of a specific message.

```
from maro.communication import Message

message = Message(tag="check_in",
                  source="worker_001",
                  destination="master",
                  body="")
```

Session Message

We provide two kinds of predefined session types for common distributed scenarios:

- **Task Session**: It is used to describe a computing task sent from master to worker. Three stages are included:
  1. The master sends the task request(s) to the worker(s);
  2. Once the worker(s) receiving the task(s), the worker(s) start to finish the task(s);
  3. The worker(s) return the computing result(s) to the master.

- **Notification Session**: It is used for information syncing and only includes two stages:
    1. The sender sends out the notification message;
    2. The receiver(s) receive the notification message.

The stages of each session are maintained internally by the proxy.

```
from maro.communication import SessionMessage, SessionType

task_message = SessionMessage(tag="sum",
                              source="master",
                              destination="worker_001",
                              body=[0, 1, 2, ...],
                              session_type=SessionType.TASK)

notification_message = SessionMessage(tag="check_out",
                                      source="worker_001",
                                      destination="master",
                                      body="",
                                      session_type=SessionType.NOTIFICATION)
```

Communication Primitives

Proxy provides a set of general-purpose communication primitives that support both blocking and non-blocking cases. These primitives are decoupled from the underlying implementation of the communication driver (protocol). The main primitives are listed below:

- `send`: Unicast. It is a blocking, one-to-one sending mode. It will watch and collect the reply message from the remote peer.
- `isend`: The non-blocking version of the `send`. A message session ID will be immediately returned, which can be used by `receive_by_id`.
- `scatter`: An advanced version of `send`. Is is used to send message(s) to peer(s) and watch and collect reply message(s) from the peer(s). `scatter` is not a real multi-cast, each message will go through the full TCP/IP stack (ZeroMQ driver). If the message you want to send is completely same and you want better performance, use the `broadcast` interface instead.
- `iscatter`: The non-blocking version of the `scatter`. The related messages session ID(s) will be returned immediately, which can be used by `receive_by_id`.
- `broadcast`: A blocking function call which is used to broadcast the message to all subscribers, it will watch and collect all subscribers' reply messages.
- `ibroadcast`: The non-blocking version of the `broadcast`. The related messages session IDs will be returned immediately, which can be used by `receive_by_id`.
- `receive`: It is used to continually receive the message.
- `receive_by_id`: It only receives the message(s) with the given session ID(s).

3.18.2 Conditional Event Register Table

The conditional event register table provides a message auto-despatching mechanism. By registering the `conditional event` and related `handler function` to the register table, the handler function will be automatically executed with the received messages when the event conditions are met.

`Conditional event` is used to declare the required message group for auto-triggering the related handler function. The unit event is the minimal component in the conditional event, it follows a three-stage format: *source*:*tag*:`amount`.

- `source`: It is used to declare the required message source. The longest-prefix matching is supported.
  - `*` is used to present any sources.
- `tag`: The `tag` attribute of the message instance.
  - `*` is used to present any tags.
- `amount`: The required message instance amount. Both a absolute integer and a relative percentage are valid for this field.
  - `%` is used to represent the relative percentages, such as 60%, 10%, etc.

```
unit_event_abs = "worker:update:10"

unit_event_rel = "worker:update:60%"
```

To support more complex business logic, we provide two operations: `AND` and `OR` to combine unit events up:

- `AND`: Valid for multiple unit events and combined unit events. The combined event condition is met if all the conditions of the sub-events are met.
- `OR`: Valid for multiple unit events and combined unit events. The combined event condition is met if any sub-event meets the condition.

```
combined_event_and = ("worker_01:update:2",
                      "worker_02:update:3",
                      "AND")

combined_event_or = ("worker_03:update:1",
                      "worker_04:update:5",
                      "OR")

combined_event_mix = (("worker_01:update:2", "worker_02:update:3", "AND"),
                      "worker_03:update:1",
                      "OR")
```

`Handler function` is a user-defined callback function that is bind to a specific conditional event. When the condition of the event is met, the related messages will be sent to the handler function for its execution.

```
# A common handler function signature
def handler(that, proxy, messages):
    """
        Conditional event handler function.

        Args:
            that: local instance reference, which needs to be operated.
```

(continues on next page)

(continued from previous page)

```
            proxy: the proxy reference for remote communication.
            messages: received messages.
    """
    pass
```

3.18.3 Distributed Decorator

Distributed decorator is a helper for generating a distributed worker class from a local functional class.

```python
from maro.communication import dist, Proxy

# Initialize proxy instance for remote communication.
proxy = Proxy(group_name="master-worker",
              component_type="worker",
              expected_peers=[("master", 1)])

# Declare the trigger condition of rollout event.
rollout_event = "master:rollout:1"

# Implement rollout event handler logic.
def on_rollout(that, proxy, messages):
    pass

# Assemble event-handler directory.
handler_dict = {rollout_event: on_rollout}

# Convert a local functional class to a distributed one.
@dist(proxy, handler_dict)
class Worker:
    def __init__(self):
        pass
```

3.19 Distributed Orchestration

MARO provides easy-to-use CLI commands to provision and manage training clusters on cloud computing service like Azure. These CLI commands can also be used to schedule the training jobs with the specified resource requirements. In MARO, all training job related components are dockerized for easy deployment and resource allocation. It provides a unified abstraction/interface for different orchestration framework see (e.g. *Grass*, *K8s*).

3.19.1 Process

The process mode is part of the *MARO CLI*, which uses multi-processes to start the training jobs in the localhost environment. To align with *Grass* and *K8s*, the process mode also uses Redis for job management. The process mode tries to simulate the operation of the real distributed cluster in localhost so that users can smoothly deploy their code to the distributed cluster. Meanwhile, through the training in the process mode, it is a cheaper way to find bugs that will happens during the real distributed training. It has the following advantages:

- Behavior is consistent with the real distributed CLI.
- Friendly to operate.

- Lightweight, no other dependencies are required.

In the Process mode:

- All jobs will be started by multi-processes and managed by *MARO Process CLI*.
- Customized settings support, such as Redis, the number of parallel running jobs, and agents check interval.
- For each job's start/stop, a ticket will be pushed into job queues. The agents monitor those job queues and start/stop job components.

Check Localhost Provisioning to get how to use it.

3.19.2 Grass

Grass is an orchestration framework developed by the MARO team. It can be confidently applied to small/middle size cluster (< 200 nodes). The design goal of Grass is to speed up the development of distributed algorithm prototypes. It has the following advantages:

- Fast deployment in a small cluster.
- Fine-grained resource management.
- Lightweight, no complex dependencies required.

Check *Grass Cluster Provisioning on Azure* and *Grass Cluster Provisioning in On-Premises Environment* to get how to use it.

Modes

We currently have two modes in Grass, and you can choose whichever you want to create a Grass cluster.

grass/azure

- Create a Grass cluster with Azure.
- With a valid Azure subscription, you can create a cluster with one command from ground zero.
- You can easily scale up/down nodes as needed, and start/stop nodes to save costs without messing up the current environment.
- Please note that the maximum number of VMs in grass/azure is limited by the available dedicated IP addresses.

grass/on-premises

- Create a Grass cluster with machines on hand.
- You can join a machine to the cluster if the machine is in the same private network as the Master.

Components

Here's the diagram of a Grass cluster with all the components tied together.

Master Components

- redis: A centralized DB for runtime data storage.
- fluentd: A centralized data collector for log collecting.
- samba-server: For file sharing within the whole cluster.
- master-agent: A daemon service for status monitoring and job scheduling.
- master-api-server: A RESTFul server for cluster management. The MARO CLI can access this server to control cluster and get cluster information in an encryption session.

Node Components

- samba-client: For file sharing.
- node-agent: A daemon service for tracking the computing resources and container statues of the node.
- node-api-server: An internal RESTFul server for node management.

Communications

Outer Environment to the Master

- The communications from outer environment to the Master is encrypted.
- Grass will use the following paths in the OuterEnv-Master communications:
  - SSH tunnel: For file transfer and script execution.
  - HTTP connection: For connection with master-api-server, use RSA+AES hybrid encryption.

Communications within the Cluster

- The communications within the cluster is not encrypted.
- Therefore, user has the responsibility to make sure all Nodes are connected within a private network and restrict external connections in the cluster.

3.19.3 K8s

MARO also supports Kubernetes (k8s) as an orchestration option. With this widely adopted framework, you can easily build up your MARO Cluster with hundreds and thousands of nodes. It has the following advantages:

- Higher durability.
- Better scalability.

We currently support the k8s/aks mode in Kubernetes, and it has the following features:

- The dockerized job component runs in Kubernetes Pod, and each Pod only hosts one component.
- All Kubernetes Pods are registered into the same virtual network using Container Network Interface(CNI).
- Azure File Service is used for file sharing in all Pods.
- Azure Container Registry is included for image management.

Check *K8S Cluster Provisioning on Azure* to see how to use it.

3.20 Dashboard Visualization

Env-dashboard is a post-experimental visualization tool, aims to provide more intuitive environment information, which will guide the design of the algorithm and continually fine-tuning.

Currently, the visualization of senario **Container Inventory Management** and **Citi Bike** are supported.

3.20.1 Dependency

Module **streamlit** and **altair** should be pre-installed.

- streamlit
- altair

Install them with:

```
pip install streamlit altair
```

3.20.2 How to Use?

Generate dumped data

The dumped data from environment is the data source of visualization. To generate data, user needs to specify the parameter **options** when creating Env object. Type of value of this parameter should be Dictionary.

If user does not need to dump data, then there is no need to pass value to this parameter. If the value for key "enable-dump-snapshot" of this parameter is an empty string, data would be dumped to the folder which start the command. If user specifies the value for key "enable-dump-snapshot" of this parameter with the path of a local file folder, and the dir_path of the local file folder does exist, data would be dumped to this folder. To be specific, each dump request would generate a data folder with a timestamp as the suffix in the local file folder.

```
opts_have_path = {"enable-dump-snapshot": "./dump_data"}

opts_have_no_path = {"enable-dump-snapshot": ""}

# dump data to folder ./dump_data.
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0",
      start_tick=0, durations=100, options=opts_have_path)

# dump data to the folder which run the command.
```

(continues on next page)

(continued from previous page)

```
env = Env(scenario="cim", topology="toy.5p_ssddd_l0.0",
      start_tick=0, durations=100, options=opts_have_no_path)
```

Data would be dumped automatically when the Env object is initialized. To get the complete reference, please view the file maro/examples/hello_world/cim/hello_streamlit.py.

For more details about Environment, please refer to Environment.

Launch Visualization Tool

To start this visualization tool, user need to input command following the format:

```
maro inspector dashboard --source_path SNAPSHOT_DUMP_DATA_FOLDER --force {true/false}
```

Parameter **force** refers to regenerate cross-epoch summary data or not, default value is 'true'. Parameter **source_path** refers to the path of dumped snapshot files. Every experiment would generate a data folder with a timestamp as the suffix, which we refer as SNAPSHOT_DUMP_DATA_FOLDER. The input of the parameter **source_path** should be the path of SNAPSHOT_DUMP_DATA_FOLDER.

Make sure that your SNAPSHOT_DUMP_DATA_FOLDER's structure is similar as following:

Folder Structure

```
./SNAPSHOT_DUMP_DATA_FOLDER
    epoch_#                          # folders to restore data of each epoch.
        {resource_holder}.csv        # attributes of current epoch.
   manifest.yml                      # basic info like scenario name, number of␣
↪epoches.
   index_name_mapping file         # relationship between an index and its name of␣
↪resource holders.
   {resource_holder}_summary.csv     # cross-epoch summary information.
```

If any file is missed compared with the expected folder structure displayed above, the command line would prompt users with an error message. The visualization tool looks for the free port to launch page in sequence, starting with port 8501. The command line would print out the selected port.

3.20.3 Feature List

Basically, each scenario has 2 parts of visualization: intra-epoch view and inter-epoch view. User could switch between them freely.

Intra-epoch view

User could view detailed information of selected resource holder or tick under this mode. In order for users to better understand the data, we separate the data into time dimension and space dimension. Users could view both the value of a resource holder's property over time and the state of all resource holders at a selected time (e.g. tick).

Content of intra-epoch view is varied between senarios. For example, in senario container_inventory_management, the attributes of resource holders are relatively complex. Thus, this view is divided into two parts: Accumulated Attributes and Detail Attributes. The former one includes the heat map of transfer volume, top-k attributes summary, accumulated attributes summary. The latter one includes the chart of two resource holders: Port and Vessel attributes in the scenario container_inventory_management. Detailed introduction please refer to Container Inventory Management Visualization.

The content of senario citi_Bike is much simpler, mainly including top-k attributes summary and the chart of resource holder: Station in senario citi_bike. Detailed introduction please refer to Citi Bike Visualization.

Epoch/Snapshot/Resource Holder Index Selection

To view the details of a resource holder or a tick, user could select the specific index of epoch/snapshot/resource holder by sliding the slider on the left side of page.

Snapshot/Resource Holder Sampling Ratio Selection

To view trends in the data, or to weed out excess information, user could select the sampling ratio of snapshot/resource holder by sliding to change the number of data to be displayed.

Formula Calculation

User could generate their own attributes by using pre-defined formulas. The results of the formula calculation could be reused as the input parameter of formula.

Inter-epoch view

User could view cross-epoch information in this view. In order to make users intuitively observe the results of the iterative algorithm, such as whether the results converge as expected, we extracted important attributes of resource holder from each epoch as a summary of the current epoch and display them centrally in this view. Users are free to choose the interval they care about and the sampling rate within the selected interval. Line chart and bar chart can effectively help users to know the results of the experiment.

Epoch Sampling Ratio Selection

To view trends in the data, or to weed out excess information, user could select the sampling ratio of epoch by sliding to change the number of data to be displayed.

Formula Calculation

Please refer to *Formula Calculation*.

3.20.4 Examples

Examples of each scenarios please refer to docs of each scenarios:

- Container Inventory Management.
- Citi Bike.

3.21 Geographic Visualization

We can use Env-geographic for both finished experiments and running experiments. For finished experiments, the local mode is enabled for users to view experimental data in order to help users to make subsequent decisions. If a running experiment is selected, the real-time mode will be launched by default, it is used to view real-time experimental data and judge the effectiveness of the model. You can also freely change to local mode for the finished epoch under real-time mode.

3.21.1 Dependency

Env-geographic's startup depends on **docker** and **docker-compose**. Therefore, users need to install docker on the machine and ensure that it can run normally. User could get docker through Docker installation.

We suggest that the version of docker should >= 1.20 and the version of docker-compose should >= 1.27.

For windows users, make sure to switch on your **Hyper-V** service to ensure docker could start successfully. Besides, users should make sure that their machine could execute **shell commands** in any directory. We suggest that windows users could install git bash to meet this requirement.

3.21.2 How to Use?

Env-geographic has 3 parts: front-end, back-end and database. Users need 2 steps to start this tool:

1. Start the database and choose an experiment to be displayed.
2. Start the front-end and back-end service with specified experiment name.

Start database

Firstly, user need to start the local database with command:

```
maro inspector geo --start database
```

After the command is executed successfully, user could view the local data with localhost:9000 by default. If the default port is occupied, user could obtain the access port of each container through the following command:

```
docker container ls
```

User could view all experiment information by SQL statement:

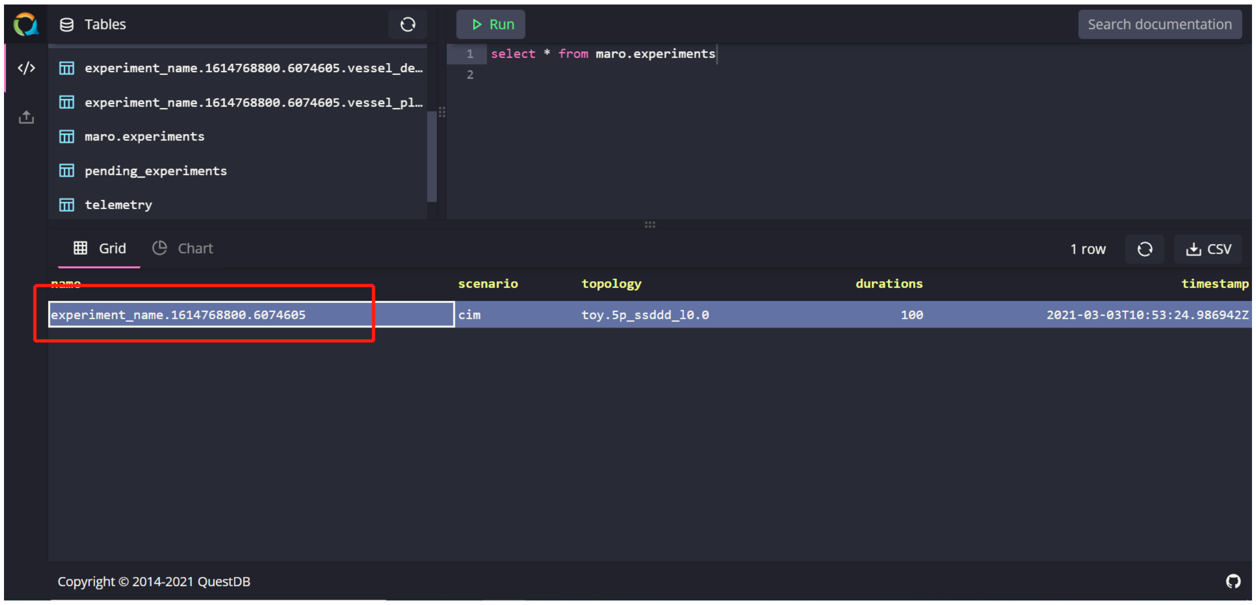
```sql
SELECT * FROM maro.experiments
```

Data is stored locally at the folder maro/maro/streamit/server/data.

Choose an existing experiment

To view the visualization of experimental data, user need to specify the name of experiment. User could choose an existing experiment or start an experiment either.

User could select a name from local database.



```
experiment_name.1614768800.6074605.vessel_de…
experiment_name.1614768800.6074605.vessel_pl…
maro.experiments
pending_experiments
telemetry
```

```sql
select * from maro.experiments
```

name	scenario	topology	durations	timestamp
experiment_name.1614768800.6074605	cim	toy.5p_ssddd_l0.0	100	2021-03-03T10:53:24.986942Z

Copyright © 2014-2021 QuestDB

Create a new experiment

Currently, users need to manually start the experiment to obtain the data required by the service.

To send data to database, there are 2 compulsory steps:

1. Set the environmental variable to enable data transmission.
2. Import relevant package and modify the code of environmental initialization to send data.

User needs to set the value of the environment variable "MARO_STREAMIT_ENABLED" to "true". If user wants to specify the experiment name, set the environment variable "MARO_STREAMIT_EXPERIMENT_NAME". If user does not set this value, a unique experiment name would be processed automatically. User could check the experiment name through database. It should be noted that when selecting a topology, user must select a topology with specific geographic information. The experimental data obtained by using topology files without geographic information cannot be used in the Env-geographic tool.

User could set the environmental variable as following example:

```
os.environ["MARO_STREAMIT_ENABLED"] = "true"

os.environ["MARO_STREAMIT_EXPERIMENT_NAME"] = "my_maro_experiment"
```

To send the experimental data by episode while the experiment is running, user needs to import the package **streamit** with following code before environment initialization:

```
# Import package streamit
from maro.streamit import streamit
# Initialize environment and send basic information of experiment to database.
env = Env(scenario="cim", topology="global_trade.22p_l0.1",
         start_tick=0, durations=100)

for ep in range(EPISODE_NUMBER):
      # Send experimental data to database by episode.
      streamit.episode(ep)
```

To get the complete reference, please view the file maro/examples/hello_world/cim/hello_geo_vis.py.

After starting the experiment, user needs to query its name in local database to make sure the experimental data is sent successfully.

Start service

To start the front-end and back-end service, user need to specify the experiment name. User could specify the port by adding the parameter "front_end_port" as following command:

```
maro inspector geo --start service --experiment_name YOUR_EXPERIMENT_NAME --front_end_
↪port 8080
```

Generally, the backend service runs on a local machine, occupying port 5000. If port 5000 is already occupied, the program would find another free port starting from 5000.

To be specific, if user would like to run the backend service in docker rather on a local machine, please run the shell script run_docker.sh under the folder maromaroclimaro_real_time_vis. It should be noted that if user run backend service with docker, data processing may be slower than local.

The program will automatically determine whether to use real-time mode or local mode according to the data status of the current experiment.

3.21.3 Feature List

For the convenience of users, Env-geographic tool implemented some features so that users can freely view experimental data.

Real-time mode and local mode

Local mode

In this mode, user could comprehend the experimental data through the geographic information and the charts on both sides. By clicking the play button in the lower left corner of the page, user could view the dynamic changes of the data in the selected time window. By hovering on geographic items and charts, more detailed information could be displayed.

The chart on the right side of the page shows the changes in the data over a period of time from the perspectives of overall, port, and vessel.

The chart on the left side of the page shows the ranking of the carrying capacity of each port and the change in carrying capacity between ports in the entire time window.

Real-time mode

The feature of real-time mode is not much different from that of local mode. The particularity of real-time mode lies in the data. The automatic playback speed of the progress bar in the front-end page is often close to the speed of the experimental data. So user could not select the time window freely in this mode.

Besides, user could change the mode by clicking. If user choose to view the local data under real-time mode, the experimental data generated so far could be displayed.

Geographic data display

In the map on the page, user can view the specific status of different resource holders at various times. Users can further understand a specific area by zooming the map. Among them, the three different status of the port: Surplus, Deficit and Balance represent the quantitative relationship between the empty container volume and the received order volume of the corresponding port at that time.

Data chart display

The ranking table on the right side of the page shows the throughput of routes and ports over a period of time. While the heat-map shows the throughput between ports over a period of time. User can hover to specific elements to view data information.

The chart on the left shows the order volume and empty container information of each port and each vessel. User can view the data of different resource holders by switching options.

In addition, user can zoom the chart to display information more clearly.

Time window selection

This feature is only valid in local mode. User can select the starting point position by sliding to select the left starting point of the time window, and view the specific data at different time.

In addition, the user can freely choose the end of the time window. When the user plays this tool, it will loop in the time window selected by the user.