Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing

USENIX ATC '23 README

Giovanni Bartolomeo[♮] Mehdi Yosofie Simon Bäurle Oliver Haluszczynski
Nitinder Mohan Jörg Ott
Technical University of Munich, Germany

[♮] `giovanni.bartolomeo@tum.de`

With this `readme`, we hope to give a detailed guide on how to install, run and test Oakestra. We'll also introduce the source code of the project, our benchmarks, and tests. The `readme` is formulated in a way that the reader can compose the Oakestra cluster the way they prefer and jump to the evaluation part directly. We remain available for any clarification or troubleshooting.

Contents

1 Introduction to the Oakestra Public Artifacts

Oakestra is an open-source project with all components publicly available on GitHub at https://github.com/oakestra. Since this is an open source project, we're continuously releasing new features and fixes. Therefore, for this `readme`, we'll refer to USENIX ATC '23 Fork.
Figure 1 shows how the components shown in our paper can be related to the repositories. Specifically, the architectural components mentioned in our paper are split into the repositories as follows.

- `oakestra/oakestra` (https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts) this repository contains the Root & Cluster orchestrators, as well as the Engines for each worker node.
- `oakestra/oakestra-net` (https://github.com/oakestra/USENIX-ATC23-Oakestra-net-Artifacts) this repository contains the Root, Cluster, and Worker network components.
- `oakestra/dashboard` (optional) (https://github.com/oakestra/dashboard) is the repository containing a front-end application that can be used to graphically interact with the platform. It's optional, and this readme will not require the installation of this component.

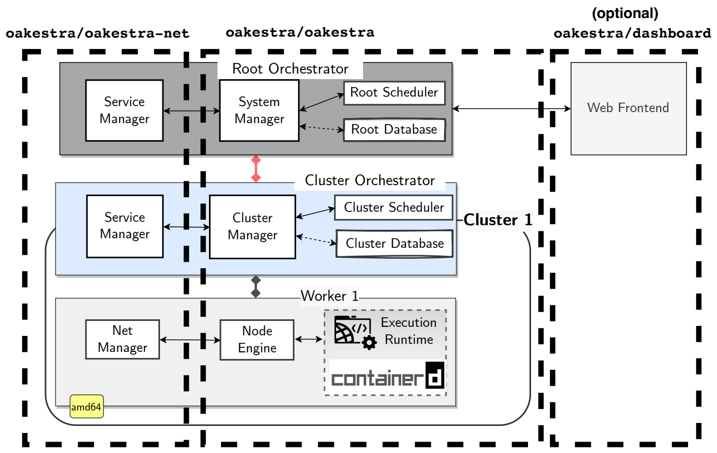


Figure 1: Summary of how the components are split across the repositories

1.1 `oakestra/oakestra`

This repository has the following file structure:

```
oakestra/oakestra
├── README.md
├── root_orchestrator/
│   ├── cloud_scheduler/
│   ├── system-manager-python/
│   ├── Experiments/
│   ├── docker-compose-amd64.yml
│   └── ...
├── cluster_orchestrator/
│   ├── cluster-manager/
│   ├── cluster-scheduler/
│   ├── docker-compose-amd64.yml
│   └── ...
├── go_node_engine/
│   └── ...
└── run-a-cluster
    ├── 1-DOC.yaml
    └── ...
```

Specifically, inside the root orchestrator, we find `system-manager-python/` and `cloud_scheduler/` that contain the *System Manager* and the *Cloud Scheduler* source code, respectively. Similarly, inside the *Cluster Orchestrator* folder, we find the source of the `cluster-manager/` and the `cluster-scheduler/`. Finally, the `go-node-engine/` contains the implementation of the *Node Engine*. The root and cluster components are implemented using `Python`, while the Node Engine is implemented in `Go` for easy integration with the runtime environments and better performance.
Both cluster and root contain a `docker-compose` file for simplifying the build and run process.

1.2 `oakestra/oakestra-net`

As discussed in Figure 1, this repository contains source code and artifacts regarding the networking components used by Oakestra.

```
oakestra/oakestra-net
├── README.md
├── root-service-manager/
│   ├── service-manager/
│   ├── docker-compose.yml
│   └── ...
├── cluster-service-manager/
│   ├── service-manager/
│   ├── docker-compose.yml
│   └── ...
└── node-net-manager/
    ├── NetManager.go
    └── ...
```

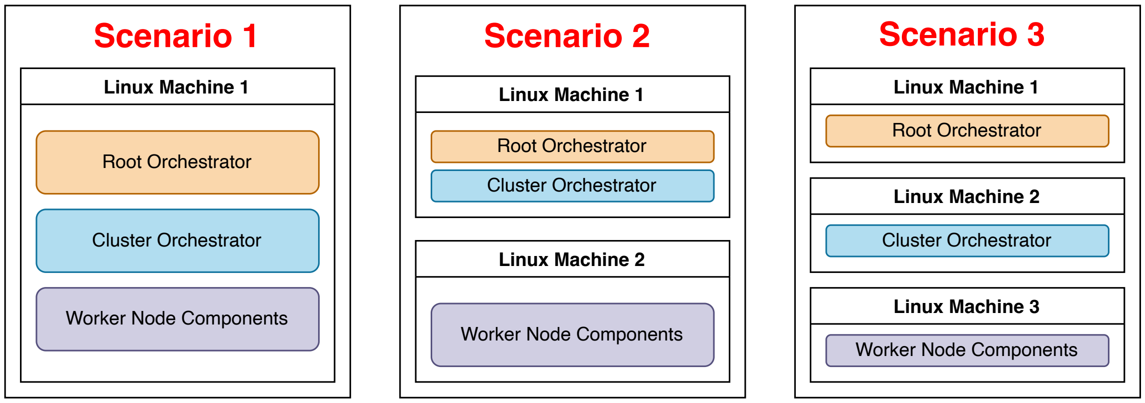
Similar to Oakestra's root orchestrator's components, the root and cluster components are developed using texttttPython, while the*Net Manager* is implemented in texttttGo. The networking stack is not mandatory in Oakestra. Without `oakestra-net`, the developer can deploy applications on the infrastructure, but the applications will be able able to carry out network related tasks. Since our experiments utilize network-capable applications, our `get started` guide (in §3) will setup both components.

2 Artifact Overview

Together with this README, we provide the scripts that we used to generate the results shown in the paper. All these artifacts can be found inside the `Experiments/` folder of `https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts`. The `Experiments/` folder will be further detailed in §5.

3 Getting Started

In this section, we'll see how to use our customized VM to start playing with Oakestra and what other possibilities we have.



(a) Single Machine Deployment (b) Master Node Deployment (c) Root-Master-Node Deployment

Figure 2: Different deployments example scenarios of Oakestra

Figure 2 shows the three distinct possibilities for setting up an Oakestra infrastructure. The most straightforward infrastructure setup is the **single cluster** and **single worker** node deployment of Oakestra. In fig. 2a, all the components are deployed inside the same machine; this setup is recommended for testing and development purposes. We'll explain how to build a custom *scenario 1* deployment in §4.2.
In fig. 2b (*scenario 2*), a worker node is an external machine (machine 2) connected to the control plane (machine 1). This setup is recommended if you want to get started with a small cluster because it enables horizontal scalability by introducing more worker nodes (on multiple Linux machines) attached to the same master (machine 1). We'll detail how to build a *scenario 2* cluster in §4.3.
Scenario 3, shown in fig. 2c, shows a single cluster setup but with separation of the Root and Cluster control plane on 2 different machines. This setup is recommended if one foresees a multi-cluster deployment in the future because new machines with new cluster orchestrators and worker nodes can be connected to the root orchestrator deployed in machine 1. We'll detail this use-case in §4.4.

4 Custom Oakestra Infrastructure

In this section, we'll dive deep into how to manually create your Oakestra cluster using your hardware. We'll start from a single node setup on §4.2, and then we'll introduce more complexity with a separation of the control plane in §4.3 and a multi-cluster setup in §4.4.

4.1 Minimum System Requirements

- Root Orchestrator
  - 2GB of RAM
  - 2 Core CPU, ARM64 or AMD64 architecture
  - 10GB disk
  - tested on: Ubuntu 20.20, Windows 10, MacOS Monterey
- Cluster Orchestrator
  - 2GB of RAM
  - 2 Core CPU, ARM64 or AMD64 architecture
  - 5GB disk
  - tested on: Ubuntu 20.20, Windows 10, MacOS Monterey
- Worker Node
  - Linux OS

- 2 Core CPU, ARM64 or AMD64 architecture
- 2GB disk
- iptables

- Network requirements
  - Root Orchestrator and Cluster orchestrator must be mutually reachable
  - Cluster orchestrator must be reachable from Worker node
  - Each worker exposes port 50103 for tunnelling
  - Root and cluster orch expose ports in the range: 10000-12000, 80

4.2 Scenario 1: Single Node Deployment

In this section, we'll deploy our first cluster with Oakestra in a single node setup, as shown in fig. 2a. To get started.

Step 1
Clone the Oakestra ATC '23 repo inside the machine that you whish to use and then move inside the folder.

```
git clone https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts.git && cd USENIX-ATC23-Oakestra
```

Step 2

Export the `CLUSTER_NAME` and `CLUSTER_LOCATION` environment variables. For this single cluster example, we'll choose simple *test* names. In general, a cluster name is a unique single-word name, while the cluster location can either be a name or geographical coordinates in the form `latitude,longitude,radius`.

```
export CLUSTER_NAME=test
export CLUSTER_LOCATION=test
```

Step 3

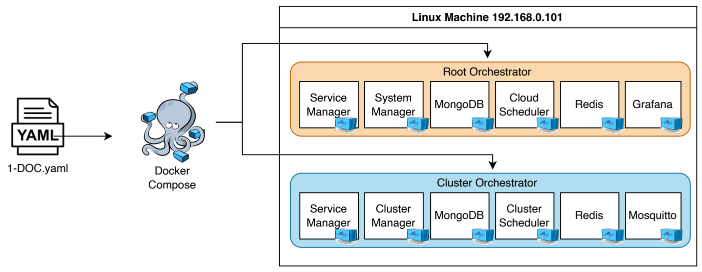


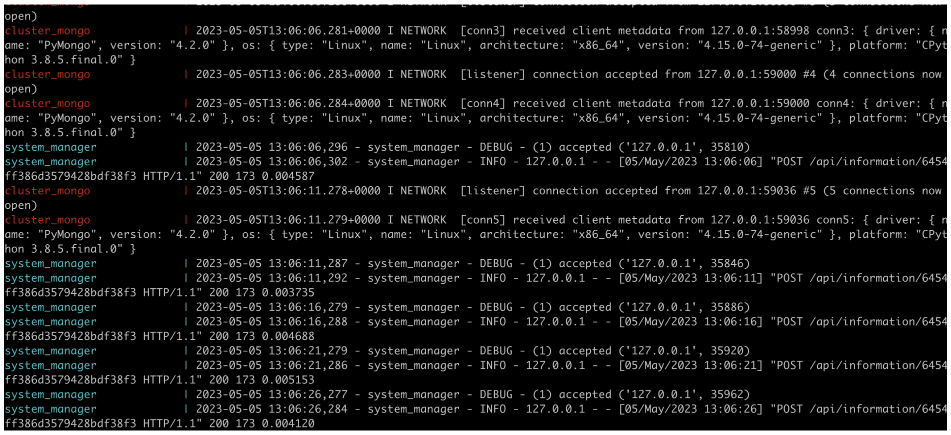
Figure 3: Components that will run after Step 3

Feed `docker-compose` with `1-DOC.yaml` file. 1-DOC stands for 1 Device One Cluster and will use docker-compose to download the latest v4.0.2 release container images to run the Root and the Cluster orchestrator on the current machine. This process is summarized in fig. 3.

```
sudo -E docker-compose -f run-a-cluster/1-DOC.yaml up
```

Disclaimer: `sudo -E` *makes sure that even if we're using* `sudo`*, docker-compose can access the environment variables exported during step 2.*

After the command pulls the release containers, creates the docker network, and startup the components, you should see something like fig. 4. Note how every 10 seconds, the `system_manager` receives a `POST /api/information` request; that's the cluster sending its aggregated information.



```
open)
cluster_mongo            | 2023-05-05T13:06:06.281+0000 I NETWORK  [conn3] received client metadata from 127.0.0.1:58998 conn3: { driver: { n
ame: "PyMongo", version: "4.2.0" }, os: { type: "Linux", name: "Linux", architecture: "x86_64", version: "4.15.0-74-generic" }, platform: "CPyt
hon 3.8.5.final.0" }
cluster_mongo            | 2023-05-05T13:06:06.283+0000 I NETWORK  [listener] connection accepted from 127.0.0.1:59000 #4 (4 connections now
open)
cluster_mongo            | 2023-05-05T13:06:06.284+0000 I NETWORK  [conn4] received client metadata from 127.0.0.1:59000 conn4: { driver: { n
ame: "PyMongo", version: "4.2.0" }, os: { type: "Linux", name: "Linux", architecture: "x86_64", version: "4.15.0-74-generic" }, platform: "CPyt
hon 3.8.5.final.0" }
system_manager           | 2023-05-05 13:06:06,296 - system_manager - DEBUG - (1) accepted ('127.0.0.1', 35810)
system_manager           | 2023-05-05 13:06:06,302 - system_manager - INFO - 127.0.0.1 - - [05/May/2023 13:06:06] "POST /api/information/6454
ff386d3579428bdf38f3 HTTP/1.1" 200 173 0.004587
cluster_mongo            | 2023-05-05T13:06:11.278+0000 I NETWORK  [listener] connection accepted from 127.0.0.1:59036 #5 (5 connections now
open)
cluster_mongo            | 2023-05-05T13:06:11.279+0000 I NETWORK  [conn5] received client metadata from 127.0.0.1:59036 conn5: { driver: { n
ame: "PyMongo", version: "4.2.0" }, os: { type: "Linux", name: "Linux", architecture: "x86_64", version: "4.15.0-74-generic" }, platform: "CPyt
hon 3.8.5.final.0" }
system_manager           | 2023-05-05 13:06:11,287 - system_manager - DEBUG - (1) accepted ('127.0.0.1', 35846)
system_manager           | 2023-05-05 13:06:11,292 - system_manager - INFO - 127.0.0.1 - - [05/May/2023 13:06:11] "POST /api/information/6454
ff386d3579428bdf38f3 HTTP/1.1" 200 173 0.003735
system_manager           | 2023-05-05 13:06:16,279 - system_manager - DEBUG - (1) accepted ('127.0.0.1', 35886)
system_manager           | 2023-05-05 13:06:16,288 - system_manager - INFO - 127.0.0.1 - - [05/May/2023 13:06:16] "POST /api/information/6454
ff386d3579428bdf38f3 HTTP/1.1" 200 173 0.004688
system_manager           | 2023-05-05 13:06:21,279 - system_manager - DEBUG - (1) accepted ('127.0.0.1', 35920)
system_manager           | 2023-05-05 13:06:21,286 - system_manager - INFO - 127.0.0.1 - - [05/May/2023 13:06:21] "POST /api/information/6454
ff386d3579428bdf38f3 HTTP/1.1" 200 173 0.005153
system_manager           | 2023-05-05 13:06:26,277 - system_manager - DEBUG - (1) accepted ('127.0.0.1', 35962)
system_manager           | 2023-05-05 13:06:26,284 - system_manager - INFO - 127.0.0.1 - - [05/May/2023 13:06:26] "POST /api/information/6454
ff386d3579428bdf38f3 HTTP/1.1" 200 173 0.004120
```

Figure 4: Schreenshot from a working cluster, just for demonstrative purposes.

Step 4

Now we need to attach a worker node to our new cluster. Open two new terminals on the same machine you used so far. We're going to need both of them during Step 7.

Note: If you're working over SSH in a remote machine, we recommend using a terminal multiplexer like TMUX, or simply creating multiple SSH sessions should be fine.

Download, untar, and install the Node Engine component.

```
wget -c https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts/releases/download/v0.4.2-atc/Node
$(dpkg --print-architecture).tar.gz && \
tar -xzf NodeEngine_$(dpkg --print-architecture).tar.gz && \
chmod +x install.sh && \
mv NodeEngine NodeEngine_$(dpkg --print-architecture) && \
./install.sh $(dpkg --print-architecture)
```

Download, untar, and install the Net Manager component.

```
wget -c https://github.com/oakestra/USENIX-ATC23-Oakestra-net-Artifacts/releases/download/v0.4.2-atc/
NetManager_$(dpkg --print-architecture).tar.gz && \
tar -xzf NetManager_$(dpkg --print-architecture).tar.gz && \
chmod +x install.sh && \
./install.sh $(dpkg --print-architecture)
```

Step 6

Startup the Net Manager and the Node Engine.

Using the first terminal available in your worker node, let's run the NetManager with local port 6000. This port will only be used by the NodeEngine.

```
sudo NetManager -p 6000
```

Then using the second terminal, let's run the NodeEngine.

```
sudo NodeEngine -n 6000 -p 10100 -a <Cluster Orchestrator Machine IP>
```

In case of success, you should see the Node Engine sending periodical messages to the cluster orchestrator.

4.2.1 Let's check if the setup is successful

Using your browser, navigate to the following URL: `http://<RootOrchestratorIP>:10000/api/docs` (In case of single node setup, scenario 1, `http://localhost:10000/api/docs`) The testable OpenAPI specification GUI of the Root Orchestrator should appear. You can use this interface to interact with Oakestra. You can deploy your app, scale it up and check your infrastructure (more details in our wiki `oakestra.io/docs`).
We'll now use this interface to check the current cluster composition. Scroll down until you see the `GET /api/clusters/active` endpoint like in fig. 5.

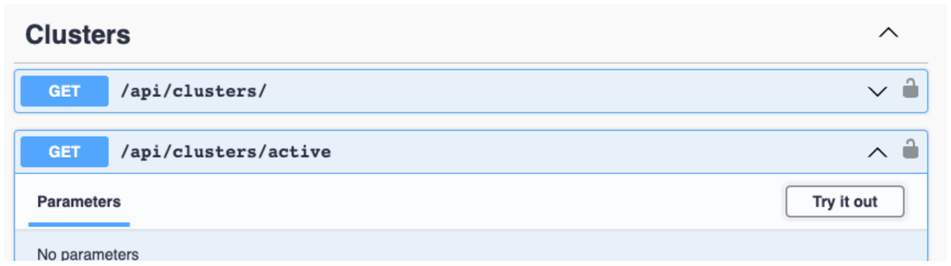
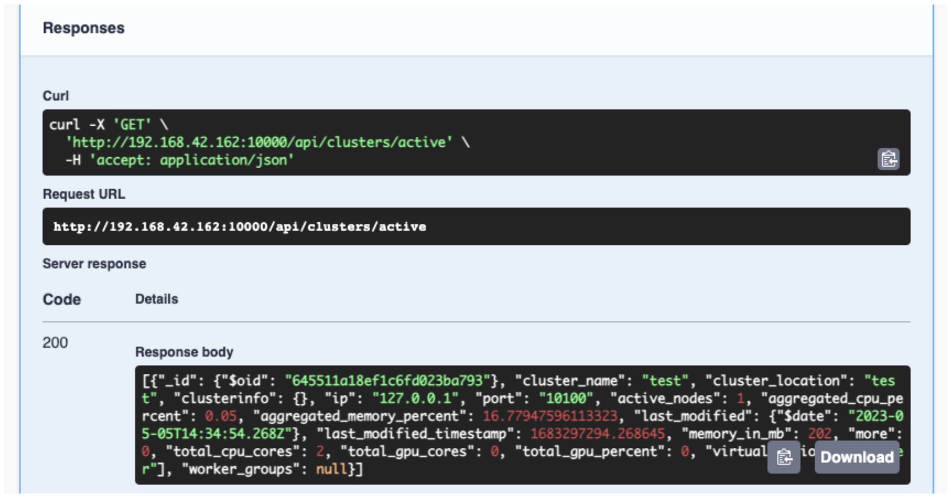


Figure 5: Schreenshot from a working cluster

After you click `try it out` button and then `execute`, you should get a response from the Root orchestrator containing a list of the clusters and the total aggregated resources for each cluster, check fig. 6.



```
curl -X 'GET' \
  'http://192.168.42.162:10000/api/clusters/active' \
  -H 'accept: application/json'
```

```
http://192.168.42.162:10000/api/clusters/active
```

200

```
[{"_id": {"$oid": "645511a18ef1c6fd023ba793"}, "cluster_name": "test", "cluster_location": "tes
t", "clusterinfo": {}, "ip": "127.0.0.1", "port": "10100", "active_nodes": 1, "aggregated_cpu_pe
rcent": 0.05, "aggregated_memory_percent": 16.77947596113323, "last_modified": {"$date": "2023-0
5-05T14:34:54.268Z"}, "last_modified_timestamp": 1683297294.268645, "memory_in_mb": 202, "more":
0, "total_cpu_cores": 2, "total_gpu_cores": 0, "total_gpu_percent": 0, "virtual
r"], "worker_groups": null}]
```

Figure 6: Schreenshot from a working cluster [specify that should look like this]

In our case, the Root orchestrator confirmed to us that we've got a Cluster called `test` with 1 worker node, 202Mb of memory and two cpu cores.

4.3 Scenario 2: Multiple Worker Nodes

In this scenario, we'll decouple the control plane of the orchestrator from the execution runtime. In particular, we'll deploy the Root and Cluster orchestrator in one node and the worker on a separate node. Please notice that you can follow the worker configuration guide on scenario 2 as many times as you like on different nodes to create multi-worker setups.

Root and Cluster orchestrator setup

Just follow Step 1 – Step 3 of §4.2.

Worker Node Setup

Please observe the following steps inside each worker node that must become part of the cluster.

Step 1

Download, untar, and install the Node Engine component.

```
wget -c https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts/releases/download/v0.4.2-atc/Node
$(dpkg --print-architecture).tar.gz && \
tar -xzf NodeEngine_$(dpkg --print-architecture).tar.gz && \
chmod +x install.sh && \
mv NodeEngine NodeEngine_$(dpkg --print-architecture) && \
./install.sh $(dpkg --print-architecture)
```

Download, untar, and install the Net Manager component.

```
wget -c https://github.com/oakestra/USENIX-ATC23-Oakestra-net-Artifacts/releases/download/v0.4.2-atc/
NetManager_$(dpkg --print-architecture).tar.gz && \
tar -xzf NetManager_$(dpkg --print-architecture).tar.gz && \
chmod +x install.sh && \
./install.sh $(dpkg --print-architecture)
```

Step 2

In this step, we'll configure the network of the networking component.
We need to edit `/etc/netmanager/netcfg.json` with the proper IP and Ports.

Disclaimer: the ownership of these files belongs to root. Don't forget sudo.

- `NodePublicAddress`: This is the network address of the current machine that is reachable from the other worker nodes of the cluster and from the cluster manager. For example, in our local setup, we use a RaspberryPi in our LAN with address `192.168.0.23`.
- `NodePublicPort`: Let's set this to 50103. This is the port used to expose the communication tunnel.
- `ClusterUrl`: the IP address where the cluster orchestrator machine is reachable starting from this node. E.g., in our local setup, we use a laptop in the same network as the RaspberryPi with local address `192.168.0.24`.
- `ClusterMqttPort`: leave this to the default 10003.

The final config file will look similar to this:

```
{
  "NodePublicAddress": "192.168.0.23",
  "NodePublicPort": "50103",
  "ClusterUrl": "192.168.0.24",
  "ClusterMqttPort": "10003"
}
```

Step 3

Startup the `NetManager` and the `NodeEngine`.

```
sudo NetManager -p 6000
sudo NodeEngine -n 6000 -p 10100 -a <Cluster Orchestrator Machine IP>
```

In case of success, you should see the Node Engine sending periodical messages to the cluster orchestrator.
You can check your setup as described in §4.2.1

4.4 Scenario 3: Multi-Cluster Setup

In this section, we'll explore how to create a setup like the one described in fig. 2c, and we'll see how to expand our setup to a multi-cluster composition.
We'll assume that the reader has already familiarized with the worker node initialization from §4.3. We'll now focus on how to install a standalone Root Orchestrator and a Standalone Cluster Orchestrator. Then, each worker node can be attached to a different Cluster Orchestrator accordingly to the cluster composition.

4.4.1 Standalone Root Orchestrator

Run the following commands inside the machine designed to be the Root Orchestrator.

```
git clone https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts.git && cd USENIX-ATC23-Oakestra

sudo -E docker-compose -f root_orchestrator/docker-compose-<arch>.yml up
```

Important: replace <arch> with `amd64` or `arm`.

4.4.2 Standalone Cluster Orchestrator

We'll now deploy as many cluster orchestrators as needed. On each cluster orchestrator machine, run the following commands.

```
export SYSTEM_MANAGER_URL=<IP ADDRESS OF THE NODE HOSTING THE ROOT ORCHESTRATOR>
export CLUSTER_NAME=<choose a name for your cluster>
export CLUSTER_LOCATION=<choose a name for the cluster's location>

git clone https://github.com/oakestra/USENIX-ATC23-Oakestra-Artifacts.git && cd USENIX-ATC23-Oakestra

sudo -E docker-compose -f cluster_orchestrator/docker-compose-<arch>.yml up
```

Important: replace <arch> with `amd64` or `arm`.

4.4.3 Throubleshooting

Error container conflict
If you have previous Oakestra installations or projects that called the containers in a similar way, it might happen that while deploying the project, you may encounter *Conflict* issues. Please remove the conflicting containers first using:

```
sudo docker container rm -f $(sudo docker container ls -a -q)
```

Docker compose error
Make sure you have docker-compose installed and that you're using the correct version. We tested this example with docker-compose v1.17.1 and docker v19.03.5.

Node Engine/Net Manager Handshake Failed
This usually happens when the Net Manager or Node Engine cannot reach the Cluster Orchestrator. Please double-check that the IP Address of the Cluster Orchestrator is properly configured with the `-a` parameter in the NodeEngine startup command and inside `/etc/netmanager/netcfg.json`. Also, make sure that the ports in the range 10000/12000 are reachable from the worker node.

Node Engine Unable to register to NetManager
Make sure that NetManager is already running on the same machine and that you're using the right startup parameter, as shown in Step 7.

Port 6000 already in use
Most like you've got another application using port 6000 or another instance of the NetManager already running. You can solve the former by repeating step 7 and replacing port 6000 with another one of your choice on each command. You can verify the latter case with the command.

```
ps -ax | grep NetManager
```

5 Experiments Artifacts

In the folder `Experiments/` of this artifact bundle, we provide the scripts that we used to obtain the results in our paper using Oakestra.

```
USENIX-ATC23-Oakestra-Artifacts/
 |__Experiments/
 |   |__Test1-deploy-undeploy/
 |   |__Test2-network-overhead/
 |   |__Test3-bandwidth/
 |   |__Test4-messages/
 |   |__Test5-stresstest/
 |   |__Test6-pipeline/
 |   |__cleanup.py
 |   |__README.pdf
 |__...
```

We provide seven test scenarios, and we'll go into detail on how to reproduce the results in the following subsections. Please note that in order to reproduce the exact behavior as represented in the paper, the same identical setup as described in the paper is necessary. Moreover, the platform heavily relies on the real-time conditions of the machine. Nevertheless, using the following scripts, it should be possible to obtain results that will be similar to the ones proposed in the paper with a fixed $\pm\delta$.

Disclaimer: In case of failures during any of this experiment, it might be related to a setup issue or some internal bug. For setup-relates issues, check the troubleshooting section of §4.2 and **??** or feel free to contact us. For any unexpected failure, we provide 2 quick lifesaver scripts.

```
python3 cleanup.py
```

This script should be executed in the root orchestrator node and allows the full undeployment of all the applications.

```
./restartworkers.sh
```

This script can be used on any worker node and restarts the NodeEngine and NetManager components. This should be used if you notice that a node becomes unresponsive.

5.1 Test 1 - Deployment Time

Setup Time	< 5 minutes
Script Execution Time	$\approx 200 - 300$ seconds

This test provides the results shown in section §5.1 of our paper. Our test exposes a web server and commands a deployment of a client for 10 consecutive times to Oakestra. The client application will perform a get request as soon as the deployment is complete, and the test web server will record the time required for the application to run and perform the request. In the `Test1-deploy-undeploy/` folder, we provide:

- Two Deployment descriptors: `client-scheduler.json` and `client-noscheduler.json`, respectively, one that requires the scheduler to pick a node and the second that explicitly requires a worker node.
- Test script: `run-test1.py` with its requirements file `requirements.txt`.
- Monitoring script: `cpu.sh`. By running this script in the background on each worker node and cluster orchestrator is possible to monitor the CPU and memory consumption. By default, it runs for 1.000 seconds exporting the CPU and memory usage of the machine every 10 seconds in a file called `cpumemoryusage.csv`.

Step 1

Move to the machine where the root orchestrator is running and copy the `Test1-deploy-undeploy/` folder. Once inside the folder, install the script requirements with:

```
pip3 install -r requirements.txt
```

(Optional) export the name of the SLA file that you want to use. If you skip this step, the default one is `client-scheduler.json`.

```
export TEST1_SLA_FILE=client-scheduler.json
```

Important: When using `client-noscheduler.json` you MUST replace HOSTNAME in `"node": "HOSTNAME"` with the hostname of the node that you want to use for your deployment. This is necessary because, to prevent the orchestrator from using the scheduling algorithm, the orchestrator must know in advance the node designed for your deployment. E.g., in our VM, we'll use `"node": "oakestra"` because that's the hostname.

Step 1.1 - (**ONLY** for Scenario 2 and 3 of fig. 2)

Export the server address. If the worker node is running on a different machine (Scenario 2 and 3 fig. 2) you also need to export the public address of the root orchestrator where you'll run the benchmark script.

```
export SERVER_ADDRESS=<Root Orchestrator Machine IP>
```

Step 3

Run the test using

```
python3 run-test1.py
```

The script will now perform 10 deployment attempts. Wait until finished.

Results

The results will be available in `results-scheduler.csv`. If everything worked successfully, you will notice how the first deployment is slower than the following ones. The first deployment takes into account the time to pull the container image from docker hub. In all our experiments, both oakestra and the other platforms, we ignored this download time.

5.2 Test 2 - Network Overhead

Setup Time	< 5 minutes
Script Execution Time	≈ 120 seconds

This experiment measures the end-to-end time required by a client application to perform 100 consecutive requests to one or more than one nginx server replica. In the `Test2-network-overhead/` folder, we provide the deployment descriptor `client-server.json` and the experiment script. This deployment descriptor creates an application with two services, `client` and `nginx`, and assigns the semantic Round Robin IP `10.30.30.30` to the nginx server. The client application exposes an endpoint that is used to start the measurement and get the results in `.csv` format. These results are used in the Networking Section of the paper. For this experiment, we provide:

- One deployment descriptor: `client-server.json`, that contains the SLA of the nginx server and the python client, respectively.
- Test script: `run-test2.py` with its requirements file `requirements.txt`.

Step 1

Copy the folder `Test2-network-overhead/` in the root orchestrator's machine and open a console window inside it. Once inside the folder, install the script requirements with:

```
pip3 install -r requirements.txt
```

Then export the number of the servers with:

```
export TEST2_SERVERS=1
```

This environment variable represents the number of server replicas to be used. The default is 1.

Step 2

Execute the test script with the following:

```
python3 run-test2.py
```

Results

Upon successful completion of the test, your console will output a CSV with the following information respectively: the request number (from 0 to 99), the experiment number (from 0 to 9), the platform name (oakestra), and the cumulative time elapsed. E.g., the last line should be similar to `99,9,oakestra,0.6984`, meaning that to complete 99 consecutive requests on iteration 9 the application needed 0.69 seconds.

Disclaimer: In case of multiple server replicas, it's possible to force oakestra to deploy them in a specific order on fixed worker nodes. This is useful to guarantee isolation and reproducible results. To enable the direct node mapping, please add to `client-server.json` a constraint of type *direct* in the nginx server listing the hostnames of the machines to be used. E.g.,

```
...
{
    "microservice_name": "nginx",
    "microservice_namespace": "test",
    "virtualization": "container",
    "memory": 50,
    ...
    "port": "6080:80/tcp",
    "addresses": {
        "rr_ip": "10.30.30.30"
    },
    "constraints": [
        {
            "type": "direct",
            "node": "hostname1;hostname2;hostname3",
            "cluster": "test"
        }
    ],
    ...
},
...
```

5.3 Test 3 - Network Bandwidth

Setup Time	< 5 minutes
Script Execution Time	∞ seconds

This experiment measures the bandwidth between two containers using the Networking Component proposed in the paper. This benchmark can be used to generate the results shown in Fig. 12 of our paper. The script will deploy 2 instances of iperf, a client and a server. The client is configured to send a file of 100MB to the server an unlimited number of times. The server is configured with the round-robin semantic IP `10.30.35.35`. After the script performs the deployment, it will output a command that can be used to attach to the container logs and check the iperf benchmark output. For this experiment, we provide:

- One deployment descriptor: `bandwidth-test.json`, that contains respectively the SLA of the iperf3 client and server.
- Test script: `run-test3.py` with its requirements file `requirements.txt`.

Step 1

Copy the folder `Test3-bandwidth/` in the root orchestrator's machine and open a console window inside it. Once inside the folder, install the script requirements with the following:

```
pip3 install -r requirements.txt
```

Step 2

Execute the test script with:

```
python3 run-test3.py
```

Wait until something similar to the following appears:

```
#### Benchmark started ####
#You can now login into the machine with IP: 192.168.42.165
#Then you can run the following command to check the realtime bandwidth:
|----------------------------------------------------------------------|
sudo ctr -n edge.io task attach iperf.tmlcj.client.test.instance.0
|----------------------------------------------------------------------|
#When you're done, press Ctrl+C to quit this script and undeploy the benchmark
```

If you're running on a single node setup (Scenario 1 of fig. 2, or the provided VM), you can run the command provided by the script directly in a new terminal window of the same machine. Otherwise, you need to SSH into the IP address provided by the script, which is one of the worker nodes that is currently executing the iperf3 client.
Important: You need to copy the command given by the script because the namespace of the application might be different.

Step 3

Execute the command provided in the previous step in the correct worker node.
Once you've collected enough results, you can go back to the test3 script and perform a cleanup by pressing `Ctrl+C`.

Results

The results of this experiment will show the bandwidth between the two container instances. By using tools like `tc` you can introduce packet loss and delay in the nodes during the experiment and directly observe the results.
Disclaimer: If not otherwise specified, the scheduler will choose the best-fit location among the nodes of your cluster. To enable the direct node mapping and explicitly choose the target machines for your deployment, please add to `bandwith-test.json` a constraint of type *direct* in both client and server services. You can list the hostnames that represent the target machines of your cluster that will be used. E.g., for the server:

```
...
{
    "microservice_name": "server",
    "microservice_namespace": "test",
    "virtualization": "container",
    "memory": 50,
    ...
    "addresses": {
        "rr_ip": "10.30.35.35"
    },
    "constraints": [
        {
            "type": "direct",
            "node": "hostname1",
            "cluster": "test"
        }
    ],
    ...
},
...
```

5.4 Test 4 - Control Messages

Setup Time	$<$ 1 minutes
Script Execution Time	$\approx$ 5 minutes

To measure the control messages, as shown in Fig. 11 of our paper, we provide a script that measures the networking system calls of the orchestrator and the worker node components. We suggest performing these measurements during Test1 operations. Inside the folder `Test4-messages/` we provide the following scripts:

- `message-counter-NetManager.sh`, can be used in any Worker Node machine to measure the messages exchanged by the networking component.
- `message-counter-NodeEngine.sh`, can be used in any Worker Node machine to measure the messages exchanged by the NodeEngine component.
- `message-counter-orchestrator.sh`, can be used in any Root or Cluster orchestrator machine to measure the messages exchanged by the orchestrators.

How To Run the Measurements

Move to the desired machine, copy the scripts, and execute

```
...
chmod +x message-counter-*
sudo ./SCRIPT-NAME.sh
...
```

Results

The results are listed respectively in the files: `strace-net-manager.txt`, `strace-node-engine.txt`, and `strace-orchestrator.txt`.

5.5 Test 5 - Stress Test

Setup Time	$<$ 5 minutes
Script Execution Time	$\approx$ 5 minutes

This test generates the results of the stress test shown in Fig. 6 of our paper. We provide a simple deployment descriptor for an nginx application and a script that deploys them. This script is meant to stress the worker and the orchestrator as much as possible. The SLA of the nginx application has 0 CPU cores and 0 MB of memory as requirements. Therefore, a node will be able to run the host's new applications until it completely crashes. For this test, we provide:

- `cpu-monitor.sh`: script that can be used to monitor and export the CPU status in real time.
- `nginx.json`: Deployment descriptor of the nginx application without any SLA requirements.
- `run-test5.py`: Python script that performs the deployment and undeployment for the stress-test.

Step 1

Copy the folder `Test5-stresstest/` in the root orchestrator's machine and open a console window inside it. Once inside the folder, install the script requirements with:

```
pip3 install -r requirements.txt
```

Step 2

Setup the number of clients that need to be deployed. The default is 10, but you can go up as much as your hardware can handle.

```
export TEST5_WORKERS_NUMBER=10
```

Disclaimer: The platform at version v0.4.2 currently has a known issue that slows down the undeployment of the applications. Therefore, after you deploy an application, it takes up to 30 seconds for the cleanup procedure of each instance. We expect to solve this issue in future releases, but as of now, the cleanup process might take a lot of time with many replicas.

Step 3

Execute the test script with:

```
python3 run-test5.py
```

Disclaimer: This procedure is meant to stress the infrastructure as much as possible. To avoid any issues, we recommend using a capable infrastructure, and we also recommend a reboot of the worker nodes afterward. For quick nodes cleanup and reboot, we provide in the root directory of the evaluation folder two scripts: `cleanup.py` that starts the undeployment of the instances and `cleanworker.sh` that, if executed in a worker node machine, restarts the worker undeploying all the workloads. The `cleanworker.sh` script is also useful to simulate worker failures.

Results

The script of this test does not give back any results directly, but the user should run the `cpu-monitor.sh` script in the background to collect the CPU and Memory consumption data on each machine.

5.6 Test 6 - Video Analytics Application Pipeline

Setup Time	≈ 5 minutes
Script Execution Time	≈ 10 minutes

Setup

This experiment deploys a video analytics pipeline as a sample payload on the platform. It can be used to generate the results shown in Fig. 9 of our paper. The pipeline consists of four services in the form of different docker containers (video source, video aggregation, object detection, and object

tracking). The video source provides an mp4 video as a reproducible RTSP network stream (typically available under port 8554). The aggregation reads the stream and forwards the frames to the detection and tracking service, respectivly. The detection service is called only for every n-th frame (30 in our case) and utilizes YOLOv3 as single-stage object detector. Object tracking keeps track of the detected objects until either (1) the object disappears or (2) the tracking algorithm is unable to keep track of the object. All code for the pipeline can be found in `https://github.com/sbaeurle/ComB`, including the CI pipeline configuration to build the corresponding docker containers yourself.
We employ the metric service provided by the ComB benchmark suite to generate the data used for the evaluation of Oakestra. The scripts and binaries provided in `Test7-ar-pipeline/` can be used to start the metric server as well as deploy the pipeline to oakestra. It consists of the following artifacts:

```
./Test7-pipeline/
├── metrics Precompiled binary of the metrics server
├── config.yaml:  Configuration of the metric endpoints employed by the pipeline
├── pipeline.json:  Oakestra deployment descriptor used to deploy the pipeline
└── run-test7.py:  Script that automatically starts the experiment
```

The experiment can be started as follows:

```
SERVER_ADDRESS=<PUBLIC IP ADDRESS OF THE ROOT ORCHESTRATOR MACHINE>
sudo chmod +x metrics
python3 run-test7.py
```

Results
The metric server generates the following folder structure:

```
./results/
└── ./<timestamp>/
    └── ./run001/
        ├── aggregation.csv
        ├── tracking.csv
        ├── detection.csv
        └── pipeline-results.csv
```

The *tracking.csv*, *detection.csv* and *aggregation.csv* contain the metrics defined in the *config.yaml* file in csv format.

Disclaimer: The pipeline automatically adjusts the output baes on the system performance. It's probable that if the resources are not enough, the benchmark will show an empty set of results.
Disclaimer: Since the pipeline was reworked since the pipeline experiments, the pipeline does not output **FPS** as metric anymore. Instead, it tries to keep up to 5 frames per second and skips any frames that could not get processed in a timely manner. However, it should still be possible to conclude similar results from the current pipeline output.