User Guide v.1.0

YAFS (Yet Another Fog Simulator) is a simulator tool based on Python of architectures such as: Fog Computing ecosystems for several analysis regarding with the placement of resources, cost deployment, network design, ... IoT environments are the most evident fact of this type of architecture.

Created: may/23/2019
Last modification: may/23/2019

using version Git: sep/12/2019

Developers:
Isaac Lera & Carlos Guerrero

Authors acknowledge financial support through grant project ORDCOT with number TIN2017-88547-P (AEI/FEDER, UE)

YAFS resources

- Public repository:
  https://github.com/acsicuib/YAFS
- Documentation:
  https://yafs.readthedocs.io/en/latest/introduction/index.html
- It’s open-published in IEEE Journal:
  https://ieeexplore.ieee.org/document/8758823
- Developed at: http://ordcot.uib.es/results/

INDEX

Installation notes:

- The explanation of the installation and the execution of the examples is addressed to novel python programmers. We encourage the use of IDEs with more flexibility to manage complex programs such as Spider or Pycharm.
- In this manual, we don’t explain the integration of YAFS with these IDE, nor give details of how to configure logging parameters, and other internal functionalities.
- We’ve not included basic commands: cd, ls, etc.
- This manual is tested on **Ubuntu 18.04.3 LTS.**

Python **2.7 is required**

You can ignore some steps according to system configuration.

Installation

Git and pip command are required.

Installing git and pip, in console:

- sudo apt install git python-pip

CONDA Installation

- ANACONDA is a distribution of python libraries. It is not necessary to install it but personally, we recommend it since ANACONDA includes several libraries very useful for analysing the YAFS results.
- **Note**: install Anaconda**2** (for python2.7)
- You can follow the official documentation to install it.
  - https://www.anaconda.com/
- or other manuals
  - https://www.digitalocean.com/community/tutorials/how-to-install-anaconda-on-ubuntu-18-04-quickstart

Cloning project

In console:

- git clone https://github.com/acsicuib/YAFS

Installing third-party libraries

The most simple way to install the dependencies is using the yaml* file from the YAFS folder. It configures a python environment.
Steps:

- (base) … /YAFS$ conda env update -f yafs.yml
- (base) … $ conda activate yafs
- (yafs) …$ *(contratulations! YAFS dependencies are installed)*

Or you can manually install the library:

- $pip install networkx (see all the packages in yafs.yml)
- etc.

*** Thanks to David Perez: https://github.com/davidperezabreu/toshare**

Your first execution

To run some `complex` python scripts is necessary to define the **working path**. We give the most simple recipe to do it: we're going to create a shell script to run python programs.

- …/YAFS/src$ **vi** run_tutorial1.sh *(you can use another editor)*

```
export PYTHONPATH=$PYTHONPATH:src/:examples/Tutorial/
python examples/Tutorial/main1.py
```

- …$chmod +x run_tutorial1.sh

- …$./run_tutorial1.sh

- **Note**: Post execution, in *src* folder, you have two new csv files with the results of the execution.

- To analyse the results, we need to understand the environment. Let's do it

Programming guide

In this section, we introduce the steps to create our IoT environment.

These steps are orientative to define the basics, but the structure of a simulation is too big to be included in these slides. Please, **Check out the main.py files in the examples/Tutorial/ folder for understanding how to combine all these pieces.**

These steps are identified in the code of those examples

Thus, we need to define:

1. the topology
2. the application/s (a composition of services)
3. the workloads or users or endpoints
4. the allocation of the services
5. the routing and orchestration of messages
6. and finally, the “deploying” in the simulator

The topology

The topology is a graph. Nodes and edges with some mandatory attributes.

https://yafs.readthedocs.io/en/latest/introduction/basic.html#network-topology

You can include more attributes

We explain two ways of create a topology:

- Json-based syntax
- NetworkX graphs constructors

Json-based topology

You load the json file with these lines:

```
t = Topology()
dataNetwork = json.load(open(‘yourNetworkDefinitionFile.json’))
t.load(dataNetwork)
```

and the syntax is:

EDGES

NODES

```
{
 "link": [
  {
   "PR": 5,
   "s": 0,
   "BW": 75000,
   "d": 2
  },
 (…)
],
 "entity": [
  {
   "RAM": 14,
   "id": 0,
   "IPT": 900
  },
(…)
]}
```

NetworkX constructors

- The *topology* class relies the management of the graph on the NetworkX library (*Topology.G*)

- NetworkX library is awesome! Thus, you can take a look several tutorials: https://networkx.github.io/

- Generating graphs with Networkx is easy https://networkx.github.io/documentation/networkx-1.10/tutorial/tutorial.html and defining the mandatory attributes as well.

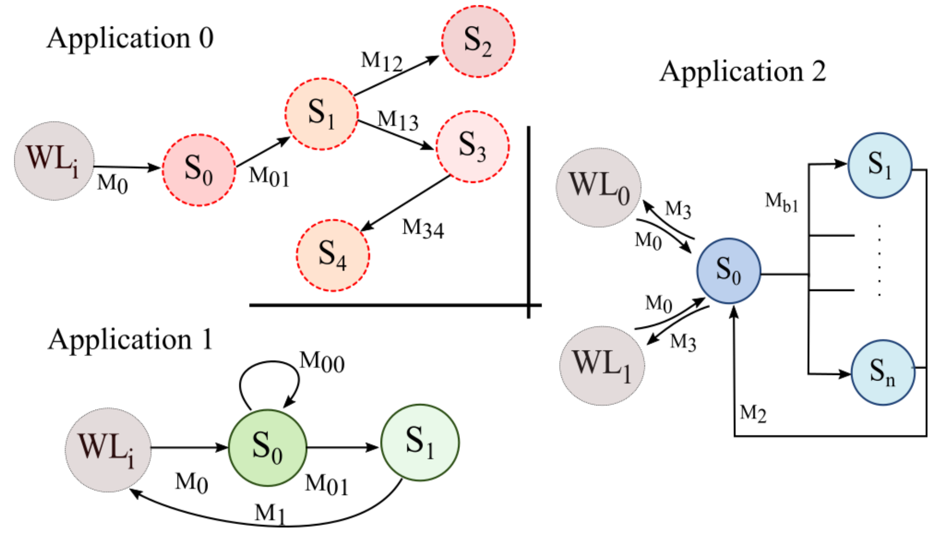
```
t = Topology()
t.G=nx.erdos_renyi_graph(100,0.15)
#for each node and edge you have to create the mandatory attributes
nx.set_node_attributes(t.G,values=valuesIPT,name="IPT")
nx.set_node_attributes(t.G,values=valuesRAM,name=“RAM")
# and the same with the edges
```

- NX manages several types of graph formats: https://networkx.github.io/documentation/stable/reference/readwrite/index.html

the Application

https://yafs.readthedocs.io/en/latest/introduction/basic.html#application

- YAFS supports the deployment of multiples applications.
- Apps are based on DAG
- You can define an app via API or using JSON-based syntax



- The API way is defined in the readthedocs and of course, multiples options are available.
- The json-based syntax is limited to a ‘simple’ exchange of messages between services. In this case, the load function transform the syntax in API calls. It means, this integration is not in the YAFS core functions,

JSON-based app definition

We use the code of *examples/MapReduceModel/exp/appDefinition.json* to describe this syntax

There are four parts: modules, messages, transmissions, and descriptive data.

The descriptive data identify the app. This information should be unique:

```
"id": 0,
"name": "0"
```

```
"id": 1,
"name": “MapReduce_1”
```

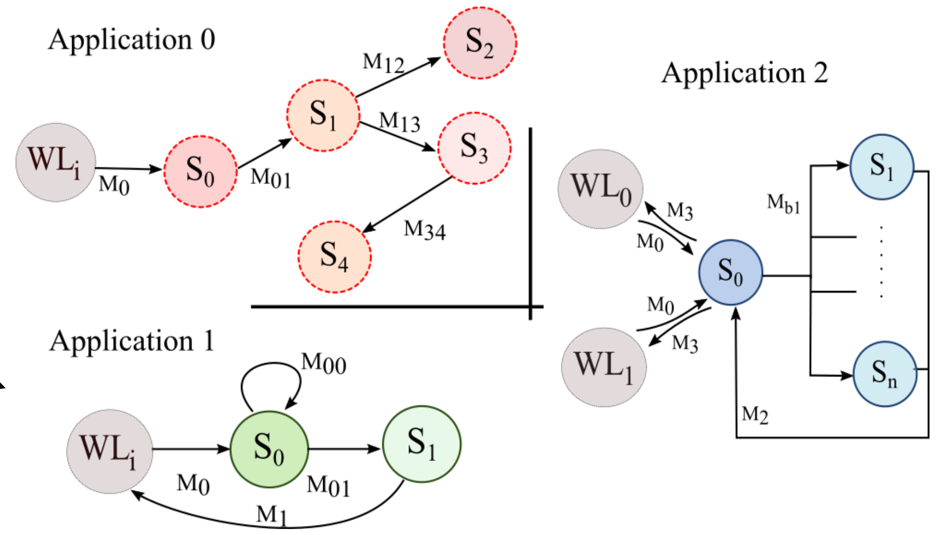
JSON-based app definition

The **modules** define the services of an app. Mandatory attributes are: id <int>, and name<str>

```
"module": [
 {
   "type": "CLOUD",
   "id": 0,
   "HD": 0,
   "name": "CLOUD_0"
 },
 {
   "type": "REPLICA",
   "id": 1,
   "HD": 2,
   "name": "0_0"
 },
 {
   "type": "REPLICA",
   "id": 2,
   "HD": 2,
   "name": "0_1"
 },
 {
   "type": "REPLICA",
   "id": 3,
   "HD": 2,
   "name": "0_2"
 }
```

App1

```
"module": [
 {
   "id": 0,
   "name": "S0"
 },
 {
   "id": 1,
   "name": "S1"
 }
```



Along this explanation, we're going to guess that the user (*WLi*) does not need the M1-message

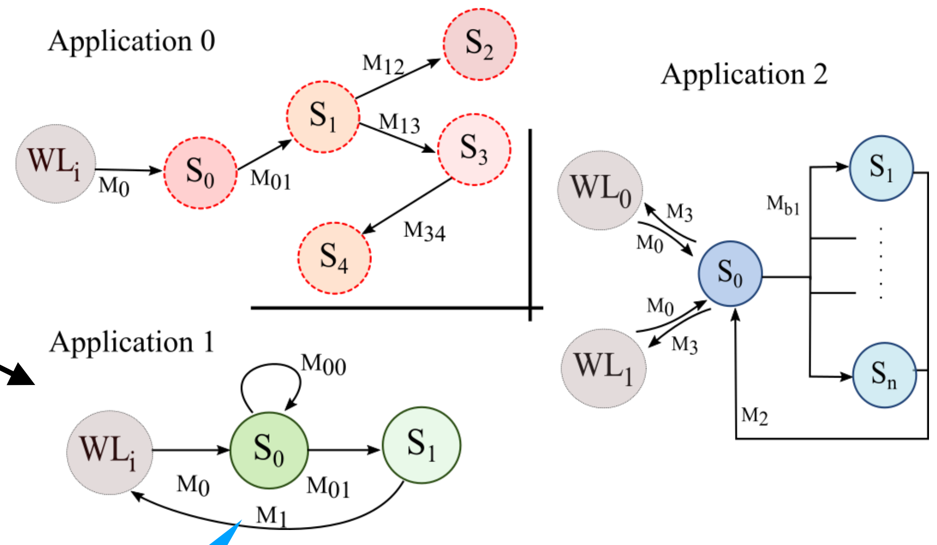
JSON-based app definition

Messages are the requests among services. Mandatory attributes are: id<int>, name<str>, s<str>, d<str>, bytes<int>, and instructions<int>

App1

```
"module": [
 {
   "id": 0,
   "name": "S0"
 },
 {
   "id": 1,
   "name": "S1"
 }
```

```
"message": [
  {
    "d": "S0",
    "bytes": 2770205,
    "name": "M.USER.APP1",
    "s": "None",
    "id": 0,
    "instructions": 0
  },
  {
    "d": "S0",
    "bytes": 2770205,
    "name": "M0_0",
    "s": "S0",
    "id": 1,
    "instructions": 20
  },
  {
    "d": "S1",
    "bytes": 22,
    "name": “M0_1”,
    "s": "S0",
    "id": 2,
    "instructions": 0
  },
```



This message (M1) has to be defined using the API or modifying the load function See: examples/Tutorial/main1.py

JSON-based app definition

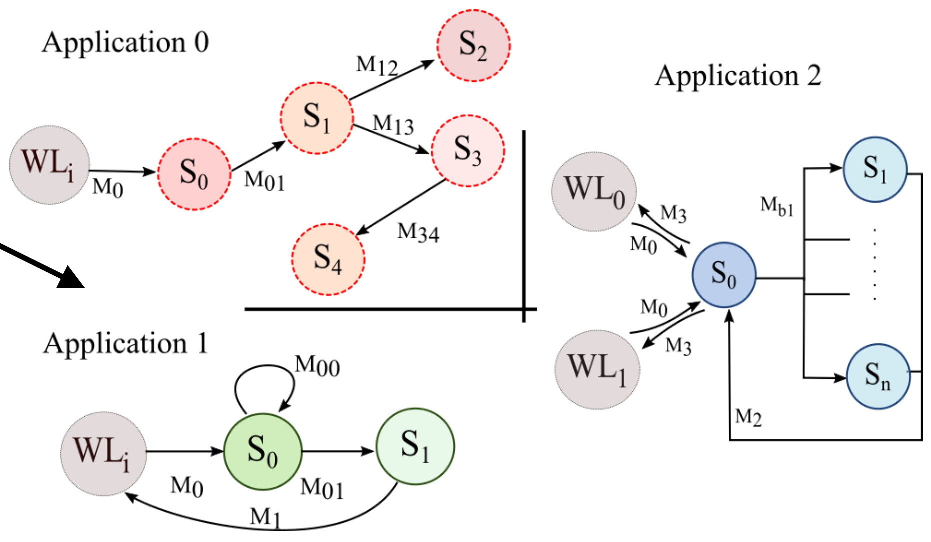
Transmissions represents how the service process the requests and generates other requests. *Fractional* is the probability to propagate the input message. A same module manages multiples messages.

```
"module": [
 {
   "id": 0,
   "name": "S0"
 },
 {
   "id": 1,
   "name": "S1"
 }
```

```
"message": [
  {
    "d": "S0",
    "bytes": 2770205,
    "name": "M.USER.APP1",
    "s": "None",
    "id": 0,
    "instructions": 0
  },
  {
    "d": "S0",
    "bytes": 2770205,
    "name": "M0_0",
    "s": "S0",
    "id": 1,
    "instructions": 20
  },
  {
    "d": "S1",
    "bytes": 22,
    "name": “M0_1”,
    "s": "S0",
    "id": 2,
    "instructions": 0
  },
```

```
"transmission": [
 {
  "module": "S0",
  "message_in": “M.USER.APP1”
 },
 {
  "message_out": “M0_0",
  "message_in": “M.USER.APP1”,
  "module": "S0",
  "fractional": 0.5
 },
 {
  "module": "S0",
  "message_in": "M0_0"
 },
 {
  "message_out": “M0_1",
  "message_in": "M.USER.APP1",
  "module": "S0",
  "fractional": 1.0
 },
 {

  "message_in": “M0_1",
  "module": "S1",
 },
],
```



JSON-based app definition

NOTE! The json-based syntax is not fully defined, so the API is still necessary to define other functionalities such as broadcasting. For this reason, the json-based function is not part of YAFS library. Therefore, if you want to use this syntax it is necessary to include / adapt /copy the function in your project: ***examples/MapReduceModel/main.py (currently the best version)***

def create_applications_from_json(data):

User / Workloads / End points

- It means, the points on the network where the initial messages of an app are generated. We called **Population**
- We can use a json-based syntax (with limited functionalities) or the API
- The simple way is using json-based syntax.

```
"""
POPULATION algorithm
"""
dataPopulation = json.load(open(pathExperiment+'usersDefinition.json'))
pop = JSONPopulation(name="Statical",json=dataPopulation) ***
```

***In some examples this function is done inside the code in the main.py file

Population json-based syntax

The syntax is simple: an array with every workload

```
{
 "sources": [
   {
     "id_resource": 20,    ←——————  An identifier of the source
       "app": “0",  ←———————————————————  the app identifier
     "message": "M.USER.APP.0", ←—————  the message name
     "lambda": 229  ←————
   },                 and the lambda value of an exponential distribution*
   {
     "id_resource": 33,
     "app": "1",
     "message": "M.USER.APP.1",
     "lambda": 223
   },
```

*the json-based syntax only works with this type of distribution but it’s easy to include the others (yafs/distribution.py)

Population json-based syntax

```
dataPopulation = json.load(open(path + 'usersDefinition.json'))
# Each application has an unique population politic
# For the original json, we filter and create a sub-list for each app politic
for aName in apps.keys():
    data = []
    for element in dataPopulation["sources"]:
        if element['app'] == aName:
            data.append(element)

    distribution = exponentialDistribution(name="Exp", lambd=random.randint(100,200), seed= int(aName)*100+it)
    pop_app = DynamicPopulation(name="Dynamic_%s" % aName, data=data, iteration=it, activation_dist=distribution)
    s.deploy_app(apps[aName], placement, pop_app, selectorPath)
```

$\longrightarrow$ **Note: each app has a deploy**

Code from: examples/MCDA/main.py

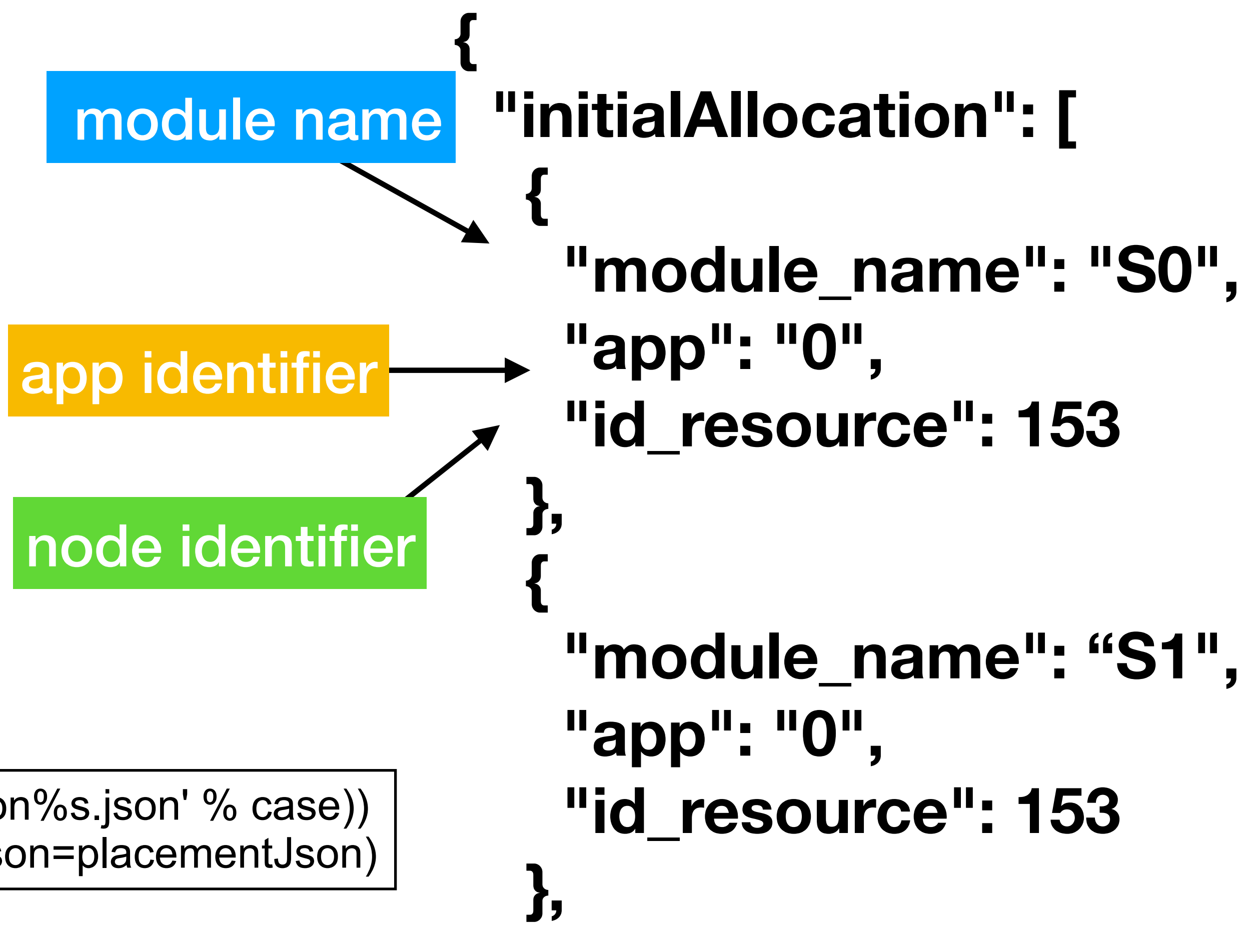
- In this case, we have different apps where users requests in different times, with a unique dynamic distributions for each app. Both **bold lines** assign a different distribution.

- **Green line** is explained in next slides since we need to explain the allocation of services and the routing message process.

Service Allocation

- The service allocation is managed by the ***Placement*** class. The placement can be defined using json-based syntax or API functions.

- The syntax is simple:



```
{
 "initialAllocation": [
  {
   "module_name": "S0",
   "app": "0",
   "id_resource": 153
  },
  {
   "module_name": “S1",
   "app": "0",
   "id_resource": 153
  },
```

- And, in the code:

```
placementJson = json.load(open(path + 'allocDefinition%s.json' % case))
placement = JSONPlacement(name="Placement", json=placementJson)
```

Message routing and orchestration

This function tries to respond to all these questions:

- What rout does take a message to reach a service? (routing problem)
- Where is this service deployed? (discovery problem)
- How many services of the same type are deployed? (scalability issues)
- What does it happen if a network link fail? or a service is unavailable in the moment that the message achieves the node where the service is deployed? (failure management)

It seems complicated but there are many examples implemented that can serve you.

Message routing and orchestration

In each project there are different files that implement all this functionalities. Let's start with the "most simple" case (example/Tutorial/main1.py)

```
"""--
SELECTOR algorithm
"""
                                 MinimunPath
#Their "selector" is actually the shortest way, there is not type of orchestration algorithm.
#This implementation is already created in selector.class,called: First_ShortestPath
selectorPath = MinimunPath()
```

MinimunPath is an extension of Selection class. The only mandatory function is **get_path**.

```
class MinimunPath(Selection):

    def get_path(self, sim, app_name, message, topology_src, alloc_DES, alloc_module, traffic,from_des):
        """
        Computes the minimun path among the source elemento of the topology and the localizations of the module

        Return the path and the identifier of the module deployed in the last element of that path
        """
        node_src = topology_src
        DES_dst = alloc_module[app_name][message.dst]

        print ("GET PATH")
        print ("\tNode _ src (id_topology): %i" %node_src)
        print ("\tRequest service: %s " %message.dst)
        print ("\tProcess serving that service: %s " %DES_dst)

        bestPath = []
        bestDES = []

        for des in DES_dst: ## In this case, there are only one deployment
            dst_node = alloc_DES[des]
            print ("\t\t Looking the path to id_node: %i" %dst_node)

            path = list(nx.shortest_path(sim.topology.G, source=node_src, target=dst_node))

            bestPath = [path]
            bestDES = [des]

        return bestPath, bestDES
```

class MinimunPath(Selection):

    def get_path(self, sim, app_name, message, topology_src, alloc_DES, alloc_module, traffic,from_des):

This function routes a message from a node to another node.

The signature of this function is:
sim is type of yafs.core (it contains all information of the simulation)
app_name is the identifier of the app implied (*redundan*t)
message: type *message*
topology_src: identifier of the node src
alloc_des: identifies the type of service/module that can dealt the message
alloc_module: internal variable of the yafs.core that contains the deployed services/modules
traffic: provides a reference of the number of messages that there are in the network (only for stats)
from_des: is the initial type of service that sends this message (only necessary in case of some failure)

```
for des in DES_dst: ## In this case, there are only one deployment
        dst_node = alloc_DES[des]
        print ("\t\t Looking the path to id_node: %i" %dst_node)

        path = list(nx.shortest_path(sim.topology.G, source=node_src, target=dst_node))

        bestPath = [path]
        bestDES = [des]

    return bestPath, bestDES
```

This function has to return an array of possible paths and an array of possible identifier of services. With returning one, it is enough. Otherwise, an empty array can be returned ([[]])

Example:
A **path** is: [86, 242, 160, 164, 130, 301, 281, 216] a list of node identifiers
A **des** is a module identifier: 23

In this case, we look for the shortest path among all nodes where deployed services are.

We can use NetworkX library algorithms !

https://networkx.github.io/documentation/stable/reference/algorithms/index.html

Message routing and orchestration

- In previous example, the shortest path function is a heavy computational task in huge networks. So, we can implement a **cache**.

- And some time, we can compute other specific characteristics of our study:

Let’s see the file: examples/MapReduceModel/selection_multipleDeploys.py

```
def get_path(self, sim, app_name, message, topology_src, alloc_DES, alloc_module, traffic, from_des):
    node_src = topology_src #entity that sends the message

    # print "Message ",message.name
    #print "Alloc DES ",alloc_DES
    DES_dst = alloc_module[app_name][message.dst] #module sw that can serve the message

    # print "Enrouting from SRC: %i  -<->- DES %s"%(node_src,DES_dst)

    #The number of nodes control the updating of the cache. If the number of nodes changes, the cache is totally cleaned.
    currentNodes = len(sim.topology.G.nodes)
    if not self.invalid_cache_value == currentNodes:
        self.invalid_cache_value = currentNodes
        self.cache = {}


    if (node_src,tuple(DES_dst)) not in self.cache.keys():
        self.cache[node_src,tuple(DES_dst)] = self.compute_DSAR(node_src, alloc_DES, sim, DES_dst,message)

    path, des = self.cache[node_src,tuple(DES_dst)]

    return [path], [des]
```

we check cache coherency considering
the number of nodes. you can use other indicator or variable

Does this “exchange” is already computed?

We look for a more complex path considering the speed of the links

Message routing and orchestration

- In case of failures you need to implement other function

Let’s see the file: examples/DynamicFailuresOnNodes/selection_multipleDeploys.py

The function get_path_from_failure is called when a node of the path is not reachable.

The path is a sequence of node identifiers [86, 242, 160, 164, 130, 301, 281, 216]

```
def get_path_from_failure(self, sim, message, link, alloc_DES, alloc_module, traffic, ctime, from_des):

    idx = message.path.index(link[0])

    if idx == len(message.path):
       return [],[]
    else:
        node_src = message.path[idx-1]
        node_dst = message.path[len(message.path)-1]

        path, des = self.get_path(sim, message.app_name, message, node_src, alloc_DES, alloc_module, traffi
                              from_des)

(…)
```

The point where the path fails

We try to find another route from a previous node

We use the same function to address the situation

Message routing and orchestration

- Perhaps, one of the more complex implementation is at:

examples/MCDA/WAPathSelectionNPlacement.py

- If there’re enough services, the algoritm deploy a new one

```
des = sim.get_DES_from_Service_In_Node(best_node,app_name,service)

logging.info("RESULTS: bestNODE: %i, DES: %s" % (best_node, des))

if des == []:
   logging.info ("NEW DEPLOYMENT IS REQUIRED in node: %i ",best_node)
   des = self.doDeploy(sim, app_name, service, best_node)
```

Note: Get_path is activated in each message request, feel free to introduce your strategies

Deploying the apps

- When we have all previous ingredients, we need to combine them, since every app has a population, a placement, and a selection.

```
#For each deployment the user - population have to contain only its specific sources
for aName in apps.keys():
    print "Deploying app: ",aName
    pop_app = JSONPopulation(name="Statical_%s"%aName,json={})
    data = []
    for element in pop.data["sources"]:
        if element['app'] == aName:
            data.append(element)
    pop_app.data["sources"]=data

    s.deploy_app(apps[aName], placement, population, selectorPath)


s.run(stop_time, test_initial_deploy=False, show_progress_monitor=False) #TEST to TRUE
```

The last line performs the simulation

Basic structure

- To sum up, your project could have the next files:

experiment / network.json
experiment / app.json
experiment / population.json
experiment / allocation.json
main.py
selector.py
jsonPopulation.py
logging.ini

Analysing the results

- A simulation generates at least two csv-based files. Attribute description is explained at:

  https://yafs.readthedocs.io/en/latest/introduction/basic.html#results

- This makes it possible to analyze any aspect of the infrastructure. For example, traffic between two nodes or between links, the latency of a particular message or set of messages, the performance of an app instead another one, etc.
  All samples have a timestamp which makes it possible to work them as a time series.

- In the projects, there are files called “analyse_results”. The analysis depends on the nature of the project. Some of them are complex than others.

- The point is that we can use libraries as Pandas.

postruning examples/Tutorial/main.py -> **Results.csv**

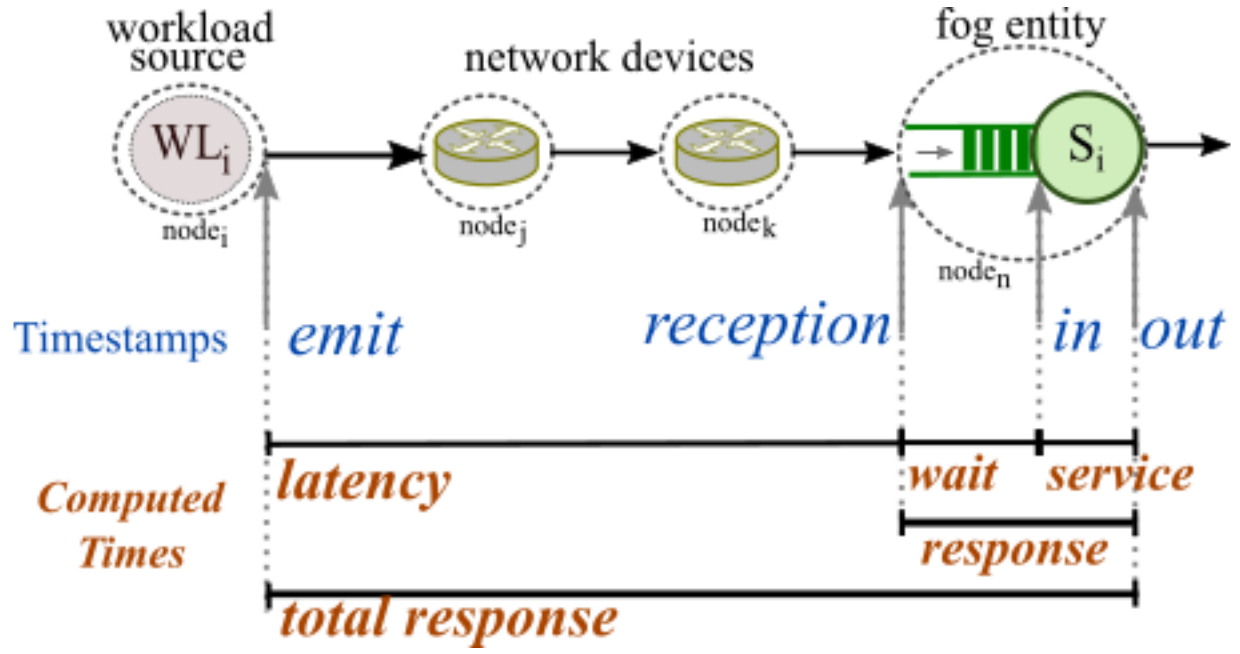


Tabla 1

id	type	app		message	DES.src	DES.dst	TOPO.src	
1	COMP_M	SimpleCase	ServiceA	M.A	0	2	1	
1	SINK_M	SimpleCase	Actuator	M.B	2	1	0	...
2	COMP_M	SimpleCase	ServiceA	M.A	0	2	1	
2	SINK_M	SimpleCase	Actuator	M.B	2	1	0	

ID-Service-source

DES.src-deployed node

ID-Service-dst

postruning examples/Tutorial/main.py -> **Results.csv**

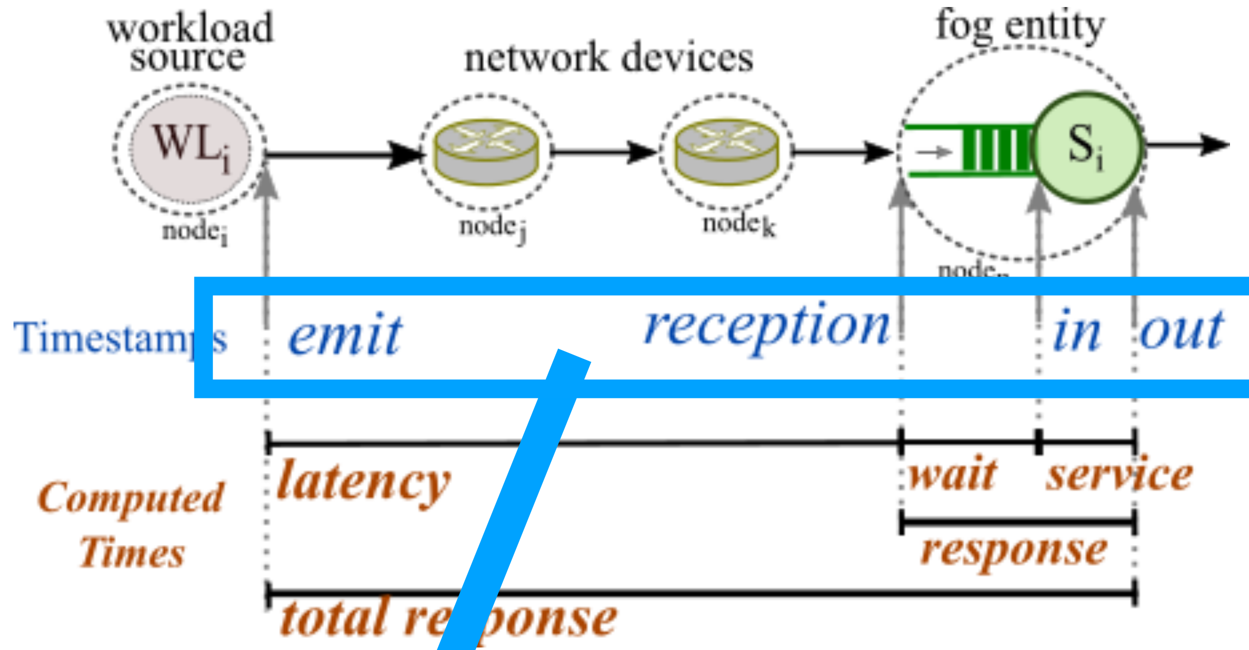


Tabla 1

id	type	app		message	DES.src	DES.dst	TOPO.src
1	COMP_M	SimpleCase	ServiceA	M.A	0	2	1
1	SINK_M	SimpleCase	Actuator	M.B	2	1	0
2	COMP_M	SimpleCase	ServiceA	M.A	0	2	1
2	SINK_M	SimpleCase	Actuator	M.B	2	1	0

DES.dst-deployed node

service time

Type

Tabla 1-1

id	TOPO.dst	module.src	service	time_in	time_out	time_emit	time_reception
1	0	Sensor	4	110.	110.0	100.0	110.001
1	2	ServiceA	0	111.005	111.00	110.0051	111.005
2	0	Sensor	4	210.001	210.0	200.0	210.001
2	2	ServiceA	0	211.00	211.0	210.00	211.0

postruning examples/Tutorial/main.py -> **Results_link.csv**

latency time between node_1 and node_0

messages waiting in the WHOLE network

simulation time

Tabla 1

id	type	src	dst	app	latency	message	ctime	size	buffer
1	LINK	1	0	SimpleCase	10.001	M.A	100	1000	0
1	LINK	0	2	SimpleCase	1.0005	M.B	110.00511950565932	500	0
2	LINK	1	0	SimpleCase	10.001	M.A	200	1000	0
2	LINK	0	2	SimpleCase	1.0005	M.B	210.00511950565934	500	0

Analysing the results

```
for it in range(nsimulations):
    fCSV = "Results_%s_%i_%i.csv"%(case,timeSimulation,it)
    df = pd.read_csv(pathSimple+fCSV)
    dtmp = df[df["module.src"]=="None"].groupby(['app','TOPO.src'])['id'].apply(list)
```

from examples/ConquestService/analyse_results.py

- In this code, we analise multiple simulations (validating the IC) -*the for*-. We load the CSV file using Pandas. In the fourth line, we obtain a list of initial requests (ids) where there is a “user” (module.src == None) and we group by app identifier and user allocation.

- **Note:** Each communication of each user has an unique *id* along all messages exchanges

Explaining the simulator: yafs.Core

- YAFS is based on a Discrete Event simulator implemented on **Simpy (another wonderful library)**

https://simpy.readthedocs.io/en/latest/

- Every transmission or message computation is an event that it is managed by Simpy processes, they are generated in the initialization of the IoT environment by a pool of MM1 queues.

- The var in Sim.env (yafs.core) contains the Simpy environment and also can control the time and the events.

  - For example, the current time can be obtained via: sim/self*.env.now*

yafs.core

- We try to comment on all the functions and variables inside yafs.core, but of course, the level of detail is never enough.

- The internal functions start with double underline "__name", these functions should be **NOT** used/modified.

- The rest of the functions generate the process to control each workload point, and the deployment of services in the network.

Creating dynamic strategies

- YAFS supports dynamic policies (population, placement, etc.). All of them can follow a distribution. Furthermore, we can create new ones.

- There are three projects to show how to implement these dynamic strategies: DynamicAllocation, DynamicFailuresOnNodes, and DynamicWorkload. Check out!

- In this brief guide, we only explain the complex way to introduce other strategies.

Creating dynamic strategies

- We use the project: *examples/ConquestService* to explain how to make it. In this project, the services change the allocations according with some rules.

- We define a class that manage these changes: **CustomStrategy**

- This strategy will start/activate following this distribution: *deterministicDistributionStartPoint*

- Besides, this class has multiple input parameters, including the “core”: **sim**

- we finally deploy this process in the simulator using: **deploy_monitor**

```
dStart = deterministicDistributionStartPoint(stop_time/2.,stop_time / 2.0 /10.0, name="Deterministic")
evol = CustomStrategy(pathResults)
s.deploy_monitor("EvolutionOfServices", evol, dStart, **{"sim": s, "routing": selectorPath,"case":case, "stop_time":stop_time, "it":it})
```

Python “__functions”

- The simulator will **only** call to the *input function*…

- Thus, **deploy_monitor only accepts functions.**
  Python functions are defined by the following signature:
  def __call__()

- **We can “call” classes defining this structure inside the class**

```
dStart = deterministicDistributionStartPoint(stop_time/2.,stop_time / 2.0 /10.0, name="Deterministic")
evol = CustomStrategy(pathResults)
s.deploy_monitor("EvolutionOfServices", evol, dStart, **{"sim": s, "routing": selectorPath,"case":case, "stop_time":stop_time, "it":it})
```

Author recommendation - Book:
Fluent Python: Clear, Concise, and Effective Programming

```
class CustomStrategy():

    def __init__(self,pathResults):
                    ...

    def __call__(self, sim, routing,case, stop_time, it):

        self.activations  +=1
        routing.invalid_cache_value = True

                    ...
```

custom parameters

The code in __call__ will be trigged in each step of the deterministicDistributionStartPoint

Future lines

- The current version of YAFS already integrates the **geo-localised** movement of workloads/users for design VoT and other types of mobile environments. This work is still in progress. There is an “example” in *examples/mobileTutorial* but it is not yet tested or documented.

- Python 2.7 has a *final countdown* (https://pythonclock.org/). Next version and future improvements in yafs.core structure will be compatible with python 3.6

References

Please use this cite to reference us:

Isaac Lera and Carlos Guerrero and Carlos Juiz YAFS: A simulator for IoT scenarios in fog computing. *IEEE Access. vol. 7, no. 1, pp. 91745-91758 (2018) https://doi.org/10.1109/ACCESS.2019.2927895 [link] [read online]*

```
@article{Lera2019yafs
author={I. {Lera} and C. {Guerrero} and C. {Juiz}},
journal={IEEE Access},
title={YAFS: A Simulator for IoT Scenarios in Fog Computing},
year={2019},
volume={7},
number={},
pages={91745-91758},
doi={10.1109/ACCESS.2019.2927895},
ISSN={2169-3536},
month={December},
}
```

Acknowledge

- YAFS is part of the project:

GOBIERNO DE ESPAÑA MINISTERIO DE ECONOMÍA, INDUSTRIA Y COMPETITIVIDAD AGENCIA ESTATAL DE INVESTIGACIÓN FEDER Fondo Europeo de Desarrollo Regional
UNIÓN EUROPEA
"Una manera de hacer Europa"

Energy and performance optimization for resource and data management in Cloud of Things with Semantic Networks. TIN2017-88547-P (MINECO/AEI/FEDER, UE).

http://ordcot.uib.es/