# YAFS Documentation

***Release 0.3***

**Isaac Lera, Carlos Guerrero**

**Sep 12, 2019**

# Contents

# CHAPTER 1

# OVERVIEW

YAFS (Yet Another Fog Simulator) is a simulation library for Cloud, Edge or Fog Computing ecosystems enabling several analysis regarding with the allocation of resources, billing management, network design, and so on.

It is a lightweight, robust and highly configurable simulator based on Simpy library (discrete event simulator) and Complex Network theory. YAFS is set by a reduced number of classes (only 7) thus we believe the learning curve is quite low compared to other similar simulators. This number of classes offer an absolute control to the user for the implementation of several customized policies and environment characteristics. We highlight the following points:

- **Topology** The infrastructure is modelled using Complex Networks theory. Any element (network devices, cloud abstractions, software modules, workloads, etc.) are represented by nodes and the links represents the possible network connection between them. In addition, Complex Networks theory provides useful topological features in order to control the deployment of services, the allocation of resources, network design considerations and other customized user policies.
- **Dynamic control** All process that the user can extend can be define dynamically, such as topology (i.e. new nodes, links failures, etc.), allocation policies, orchestration, etc.
- **Request evolution** Service requests in FOG environments is not always reduced to the same access points along the whole simulation. Requests can be generated from any point of the network following a temporary distribution.
- **Placement algorithm** Yet another classical module that decides how to assign module applications to the topology.
- **Selection algorithm** In a network, routing can be controlled by network devices but with new Fog applications the applications can controlled these messages, it depends on the user abstraction level. It offers new analytical models for the adaptation of traffic.
- **Customized distribution** User can generate events to control policies or whatever action in the simulator using customized distributions as for example a simple array of timestamps to deploy software modules.

YAFS gathers the main events in a raw format. There is not hidden variables or *stranger things* where this data is stored. This data can be accessed from any point of the simulator so any module has access to the stats.

The documentation contains a *tutorial*, *architecture details* explaining key concepts, a number of *examples* and the *API reference*.

YAFS is released under the MIT License.

## 1.1 Installation

YAFS requires Python 2.7 (Python 3.6 or above is not supported)

You can download and install YAFS manually:

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

## 1.2 Cite this work

Please, consider including this reference in your works or publications:

```
Isaac Lera, Carlos Guerrero, Carlos Juiz. YAFS: A simulator for IoT scenarios in fog␣
↪computing. IEEE Access. Vol. 7(1), pages 91745-91758,
10.1109/ACCESS.2019.2927895, Jul 10 2019.

https://ieeexplore.ieee.org/document/8758823

@ARTICLE{8758823,
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},
keywords={Relays;Large scale integration;Wireless communication;OFDM;Interference␣
↪cancellation;Channel estimation;Real-time systems;Complex networks;fog computing;
↪Internet of Things;simulator},
doi={10.1109/ACCESS.2019.2927895},
ISSN={2169-3536},
month={},
```

Please let it knows us if you use this project in your research. We will cite them. Thank you

You can find other related works developed with YAFS in the readme of YAFS git hub repository.

CHAPTER 2

# Acknowledge

Authors acknowledge financial support through project ORD-CoT (TIN2017-88547-P MINECO, SPAIN).

# CHAPTER 3

# YAFS in 5 Minutes

In this section, you'll learn the basics of YAFS in just a few minutes.

## 3.1 Installation

YAFS is implemented in pure Python and has some dependencies. YAFS runs on Python 2 (>=2.7). PyPy is also supported. If you have pip installed, just type

```
$ pip install yafs
```

### 3.1.1 YAFS dependencies

- NetworkX (2.0) for topology management
- Simpy (3.0.10) implements the discrete-event simulator
- Pandas (0.18.0) is used to analyses the results

You can use whatever python manager package for the installation of these libraries

We want to thank to the software community of these projects for their enormous work as well as the authors et al. of Simpy simulator. YAFS efficiency relies mainly on Simpy project.

## 3.2 Glossary of terms

Below there is a list of the main concepts and equivalences with terms of other works. To deepen in them, it is necessary to resort to related works in this field.

- **Module** is a part of the application that it can be deployed in a computational or any other type of device.
- **Entity**, is the abstraction of any type of device

- **Link** a network connection between two entities
- **Pure Source** is a special type of node who only generates messages (i.e. sensor devices)
- **Pure Sink** is a special type of node who only receives messages (i.e. actuator devices)
- **Placement**, this module controls the allocation of application module
- **Message** is the representation of a network package that it enables the action of a application module.

## 3.3 Basic Concepts

For running a simulation, you must define the following elements (we have used a basic example to explain them tutorial/main1.py) :

### 3.3.1 Network Topology

Firstly, the definition of a topology. A topology consists of a set of nodes and links. The nodes are elements of the network that serve as a link between other elements and have the ability to execute or host the application.

That is, a node has associated computing characteristics:

- **IPT** Instructions per simulation time. It is equivalent to IPS but it is the user how fix the unit time (i.e. seconds, milliseconds, etc.). Time simulation units is set by the user.
- **RAM** Memory available
- **COST** Cost per unit time

A link has associated these performance characteristics:

- **BW** Channel Bandwidth: in Bytes
- **PR** Channel Propagation speed
- The **Latency** is dynamically computed using: (Message.size.bits / BW) + PR)

A network can be created using a dictionary structure (or json file) or through some implemented algorithms from specific libraries compatibles with Networkx (see *architecture details*).

In the definition of your devices, nodes, you can include your custom tags.

```
from yafs.topology import Topology

topology_json = {}
topology_json["entity"] = []
topology_json["link"] = []

cloud_dev    = {"id": 0, "model": "cloud","mytag":"cloud", "IPT": 5000 * 10 ^ 6, "RAM
↪": 40000,"COST": 3,"WATT":20.0}
sensor_dev   = {"id": 1, "model": "sensor-device", "IPT": 100* 10 ^ 6, "RAM": 4000,
↪"COST": 3,"WATT":40.0}
actuator_dev = {"id": 2, "model": "actuator-device", "IPT": 100 * 10 ^ 6, "RAM": 4000,
↪"COST": 3, "WATT": 40.0}

link1 = {"s": 0, "d": 1, "BW": 1, "PR": 10}
link2 = {"s": 0, "d": 2, "BW": 1, "PR": 1}

topology_json["entity"].append(cloud_dev)
```

(continues on next page)

(continued from previous page)

```
topology_json["entity"].append(sensor_dev)
topology_json["entity"].append(actuator_dev)
topology_json["link"].append(link1)
topology_json["link"].append(link2)

t = Topology()
t.load(topology_json)
```

### 3.3.2 Application

Secondly, the application follows the DDF model, in which an application is modeled as directed graph. The vertices of the directed acyclic graph (DAG) representing modules that perform processing on incoming data and edge denoting data dependencies among modules.

An application is set by a group of modules. A module can create messages (a pure source / sensor), a modul can consume messages (a pure sink / actuator ) and other modules can do both tasks.

The dependencies among modules are defined through messages.

A **message** is the representation of a request between two modules. A message is set by an unique ID in the same application. It has a source and destiny module, a length of instructions, and a number of bytes. If the message arrive a pure sink this message does not require the last two attributes. In YAFS, it is possible to define *broadcast messages*.

```
m_a = Message("M.A", "Sensor", "ServiceA", instructions=20*10^6, bytes=1000)
m_b = Message("M.B", "ServiceA", "Actuator", instructions=30*10^6, bytes=500)
```

We illustrate this example using *EEG_GAME application*[1] following the idea used in CloudSim[2] and other similar cloud simulators.

---

**Note:** User defines the distribution functions that modules use to create and send the different messages. In this example, the user defines: *fractional_selectivity* and *next_time_periodic*.

---

---

**Note:** In *sim.utils* package there are defined several distributions

---

```
import random
from yafs.application import Application
from yafs.application import Message


def create_application():
    # APLICATION
    a = Application(name="SimpleCase")

    # (S) --> (ServiceA) --> (A)
    a.set_modules([{"Sensor":{"Type":Application.TYPE_SOURCE}},
                   {"ServiceA": {"RAM": 10, "Type": Application.TYPE_MODULE}},
                   {"Actuator": {"Type": Application.TYPE_SINK}}
```

(continues on next page)

[1] Gupta, H., Vahid Dastjerdi, A., Ghosh, S. K., & Buyya, R. (2017). iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. Software: Practice and Experience, 47(9), 1275-1296.

[2] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., & Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and experience, 41(1), 23-50.

(continued from previous page)

```
                    ])
    """
    Messages among MODULES (AppEdge in iFogSim)
    """
    m_a = Message("M.A", "Sensor", "ServiceA", instructions=20*10^6, bytes=1000)
    m_b = Message("M.B", "ServiceA", "Actuator", instructions=30*10^6, bytes=500)

    """
    Defining which messages will be dynamically generated # the generation is␣
↪controlled by Population algorithm
    """
    a.add_source_messages(m_a)

    """
    MODULES/SERVICES: Definition of Generators and Consumers (AppEdges and␣
↪TupleMappings in iFogSim)
    """
    # MODULE SERVICES
    a.add_service_module("ServiceA", m_a, m_b, fractional_selectivity, threshold=1.0)

    return a

app1 = create_aplication("Tutorial1")
```

### 3.3.3 Population model

In real scenarios, sensors can move in the IoT ecosystem which it means, they can invoke services for several access points. For that reason, each population model is associated to one application and it runs according to a set of events or time distribution.

The association of controls to a node can be dynamic and this process is executed every certain time to change the creation policies associated with the nodes. The population model can be so complex as you wish. More advanced details in *architecture details*.

The most simple case, it is a statical creation of requests. For each message of a pure service source (or sensor), it will have a generation control associated to it.

```
from yafs.population import Statical

pop = Statical("Statical")
pop.set_src_control({"model": "sensor-device", "number":1,"message": app.get_message(
↪"M.A"), "distribution": deterministicDistribution,"param": {"time_shift": 100}})#5.
↪1}})
pop.set_sink_control({"model": "actuator-device","number":1,"module":app.get_sink_
↪modules()})
```

### 3.3.4 Selector model

This module is the "service orchestration". It is the coordination and arrangement of multiple services exposed as a single aggregate service.

The most simple case is one to one module using the shortest path between both modules.

```
selectorPath = MinimunPath()
```

where internally this function is customized by the user. Mandatory returns are two arrays ( bestpath and bestdes). The first one is an array among id-nodes of the topology. The second one is also a sequence of id-process who are deployed in that nodes.

```
class MinimunPath(Selection):

    def get_path(self, sim, app_name, message, topology_src, alloc_DES, alloc_module,␣
↪traffic):
        """
        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
```

### 3.3.5 Placement algorithm

In this module is implemented the algorithm to allocation resources. Placement module has two public methods: *initial_allocation* and *run*. The first is used in the first deployment of the application, the second one is used to move this modules in the topological entities according with some multi-objetive allocation algorithm or whatever idea.

The most usual case is a deployment in the cluster. In this case, sink nodes (actuators) are fixed in the topology.

```
placement = CloudPlacement("onCloud") # it defines the deployed rules: module-device
placement.scaleService({"ServiceA": 1})
```

where: .. code-block:: python

    from yafs.placement import Placement

    class CloudPlacement(Placement):

**def initial_allocation(self, sim, app_name):** #We find the ID-nodo/resource value = {"my-tag": "cloud"} # or whatever tag

id_cluster = sim.topology.find_IDs(value) app = sim.apps[app_name] services = app.services

**for module in services:**

**if module in self.scaleServices:**

**for rep in range(0, self.scaleServices[module]):** idDES = sim.deploy_module(app_name,module,services[module],id_cluster)

### 3.3.6 Running the simulator

Once defined the previous elements, we can associate them to the simulator. To do this, you have to *deploy* the application with its respective topology policies. Once this is done, we can launch the simulation.

```
s = Sim(t) # t is the topology
simulation_time = 100000
s.deploy_app(app, placement, pop, selectorPath)
s.run(simulation_time,show_progress_monitor=False)
```

### 3.3.7 Results

The results are stored in a csv format in two files. One of the main events is the registration of each message in each entity of the topology that manages it. In these types of events, the following attributes are recorded:

- **type** It represent the entity who run the taks: a module (COMP_M) or an actuator (SINK_M)
- **app** application name
- **module** Module or service who manages it
- **service** service time
- **message** message name
- **DES.src** DES process who send this message
- **DES.dst** DES process who receive this message (the previous module(
- **TOPO.src** ID node topology where the DES.src module is deployed
- **TOPO.dst** ID node topology where the DES.dst module is deployed
- **module.src** the module or service who send this message
- **service** service time
- **time_in** time when the *module* accepts it
- **time_out** time when the *module* finishes its process
- **time_emit** time when the message was sent
- **time_reception** time when the message is accepted by the module

---

**Note:** The **units of time** have the scale defined by the user in the respective distribution units. It means, the results of time are the times of the simulator.

---

```
type,app,module,message,DES.src,DES.dst,TOPO.src,TOPO.dst,module.src,service,time_in,
↪time_out,time_emit,time_reception
COMP_M,SimpleCase,ServiceA,M.A,0,2,1,0,Sensor,0.004119505659320882,110.008,110.
↪01211950565931,100.0,110.008
SINK_M,SimpleCase,Actuator,M.B,2,1,0,2,ServiceA,0,111.01611950565932,111.
↪01611950565932,110.01211950565931,111.01611950565932
COMP_M,SimpleCase,ServiceA,M.A,0,2,1,0,Sensor,0.004119505659320882,210.008,210.
↪01211950565934,200.0,210.008
SINK_M,SimpleCase,Actuator,M.B,2,1,0,2,ServiceA,0,211.01611950565933,211.
↪01611950565933,210.01211950565934,211.01611950565933
COMP_M,SimpleCase,ServiceA,M.A,0,2,1,0,Sensor,0.004119505659320882,310.008,310.
↪0121195056593,300.0,310.008
SINK_M,SimpleCase,Actuator,M.B,2,1,0,2,ServiceA,0,311.01611950565933,311.
↪01611950565933,310.0121195056593,311.01611950565933
COMP_M,SimpleCase,ServiceA,M.A,0,2,1,0,Sensor,0.004119505659320882,410.008,410.
↪0121195056593,400.0,410.008
```

The other file storages the transmission process in the network

```
type,src,dst,app,latency,message,ctime,size,buffer
LINK,1,0,SimpleCase,10.008,M.A,100,1000,0
LINK,0,2,SimpleCase,1.004,M.B,110.01211950565931,500,0
LINK,1,0,SimpleCase,10.008,M.A,200,1000,0
LINK,0,2,SimpleCase,1.004,M.B,210.01211950565934,500,0
LINK,1,0,SimpleCase,10.008,M.A,300,1000,0
LINK,0,2,SimpleCase,1.004,M.B,310.0121195056593,500,0
```

In these types of events, the following attributes are recorded:

- **type** Link type
- **src** Source of the message - ID node topology
- **dst** Destination of the message - ID node topology
- **app** application name
- **latency** the time taken to transmit the message between both nodes.
- **message** message name
- **ctime** simulation time
- **size** size of the message
- **buffer** This variable represents the number of waiting messages in all the links.

How to obtain statistics of these results depends on the user, although there are a good number of processes implemented to obtain the main stats such as: average response time, average service time, average wait time, node utilization, average link latency, costs, and so on.

You can find this example in the following subsection: *basic example*.

So far, we have explained the main parts of the simulator. Maybe it takes more than 5 minutes to understand this modelling, but the title of the section was attractive with that number. If you want to go deeper, you have to look at the rest of the sections: *architecture details* explaining key concepts, a number of *examples* and the *API reference*.

CHAPTER 4

# Architecture

Architecture definition is at 0% of progress. :( sorry

## 4.1 Architecture

TODO

# CHAPTER 5

# Examples

Tutorial example is at 50% of progress.

## 5.1 Tutorial Example

In this section, we explain the examples from Tutorial/ directory but you can see most sophisticated implemented cases:

- **Tutorial/main1.py** The most basic case: cloud allocation, and minimum path routing,
- **Tutorial/main2.py** Using first case, we scale the modules in the cloud and we use a round robin scheduler.
- **Tutorial/main3.py** Using second case, we show how to implement a dynamic control
- **VRGameFog-IFogSim-WL/main.py** This implemention have been used to compare YAFS simulator with iFogSim, implementing a close setup of the experiments.
- **FogCentrality/experiment1.py** In this case, we have analysed how the topology and the allocation of modules affects in the latency time.
- **FogCentrality/experiment2.py**
- **DynamicAllocation/.py** It is a demo of how to implement a dynamic allocation of modules according a customized distribution. We use a random euclidean network from a Graphml format, and several selection, population, and allocation implementations.
- **DynamicFailuresOnNodes/main.py** From a euclidean random network we dynamically remove nodes.
- **DynamicWorkload/main.py** In this case, we simulate the movement of users in the network. In each step of the customized distribution nodes are allocated in the next node of the path to reach the point on all modules are.

TODO Explain with detail the first three cases.

CHAPTER 6

# API REFERENCE

The API reference provides detailed descriptions of YAFS's classes and functions. It should be helpful if you plan to extend YAFS with custom components.

## 6.1 yafs

## 6.2 yafs.core — Simulator engine

## 6.3 yafs.topology

## 6.4 yafs.application

## 6.5 yafs.population

## 6.6 yafs.placement

## 6.7 yafs.selection

## 6.8 yafs.distribution

## 6.9 yafs.stats

## 6.10 yafs.metrics

## 6.11 yafs.utils

CHAPTER 7

# ABOUT

YAFS is designed and implemented by Isaac Lera and Carlos Guerrero. Both work at Computer Science Faculty in the University Of Balearic Islands.

CHAPTER 8

# Indices and tables

- genindex
- search