User Guide

CupCarbon Twin IoT

Dr Ahcène Bounceur

Version 1.02

Introduction

CupCarbon Twin IoT (Version 6) is the latest version of CupCarbon.It is developed mainly to design Digital Twins and any other project using IoT nodes. It is designed for creating Digital Twins and various IoT node-based projects. CupCarbon Twin IoT provides the possibility to incorporate nodes capable of executing a wide range of system-based programs, such as Python, C/C++, NodeJS, Julia, exe, and more. Subsequently, through the program's print command, seamless communication with CupCarbon is established, enabling actions on the Graphical User Interface. This includes tasks like marking/unmarking and moving nodes, displaying messages, and more. Additionally, integration of both real and virtual IoT nodes within the same project environment can be achieved through the MQTT protocol.

The updated version supersedes Version 5, which has been deprecated. The rationale behind this decision is that Version 5 relies on the Jython library, which limits program flexibility. Due to its Java-based foundation, incorporating external libraries is not feasible. Essentially, in Jython, the program is authored in Python but executed in a Java environment. If one intends to include external libraries in a Jython program, it is advisable to do so in Java.

In the new CupCarbon Twin IoT, however, a significant advancement is realized. It permits the use of a completely independent Python environment or even any other programming language of choice. This heightened flexibility enhances the capabilities of the system.

Finally, for the WSN simulation, refer to the previous User Guide which can be downloaded from the CupCarbon.com website or via the following link:

http://cupcarbon.com/cupcarbon_ug.html

Start with CupCarbon

The following webpage presents all the elements required to start using CupCarbon. It is required to install Python and Paho (MQTT). If necessary, you can install NodeJS, Julia, etc.

http://cupcarbon.com/CupCarbon-Tutorials.html

The environment

The new environment of CupCarbon is almost the same as the previous ones, except the possibility to add IoT nodes. An IoT node is represented by a picture and it can be marked, unmarked and moved. Figure 1 shows some IoT nodes added in CupCarbon. To change the picture of an IoT node, select it and click on the lowercase j key.

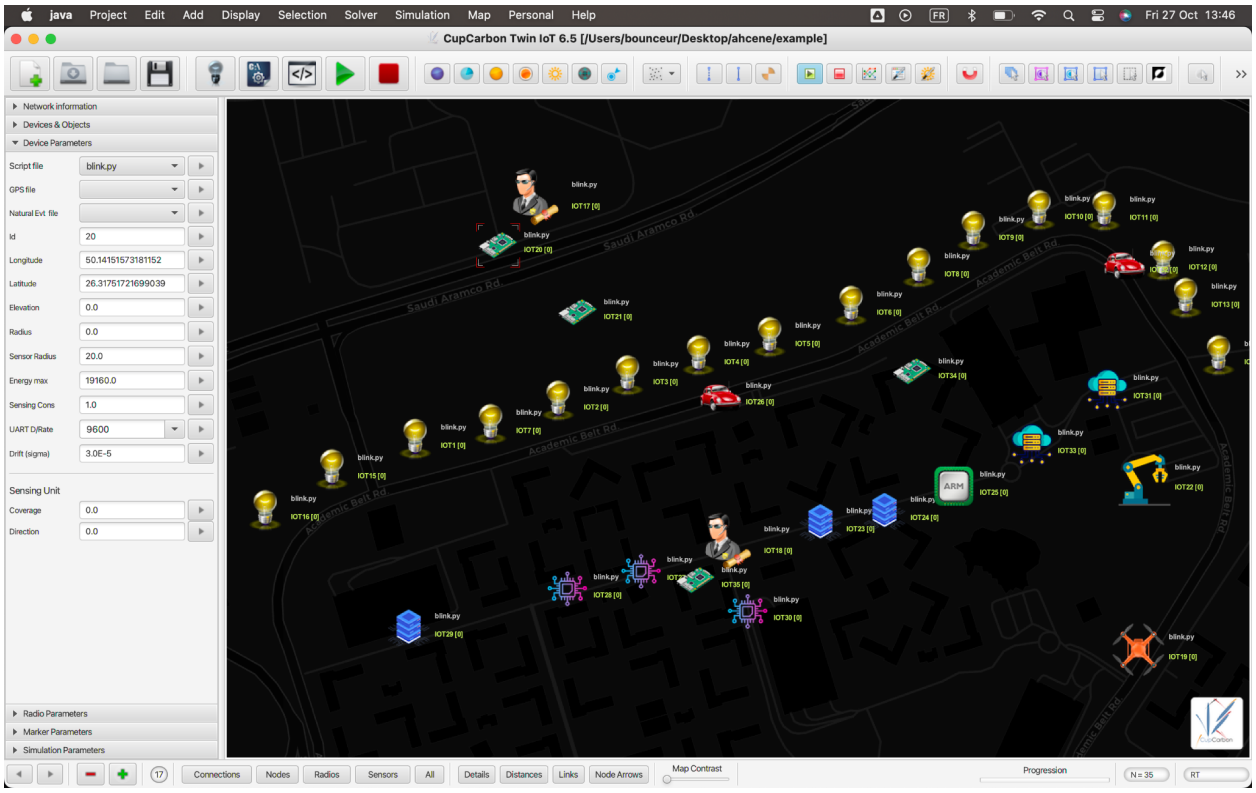

Figure 1. CupCarbon environment

The toolbar to run IoT simulation is shown by Figure 2. It contains 5 buttons.

Figure 2. Toolbar of the IoT simulator.

The role of each button is given as follows:

Button 1: add IoT node

This button allows you to add an IoT on the Map of CupCarbon. Once the node is added, it is possible to change its image by selecting it and clicking on the key j.

Button 2: define the paths of the execution commands

This button allows you to define the different paths or just the command used to run your programs. It opens the following window allowing us to define 4 paths.

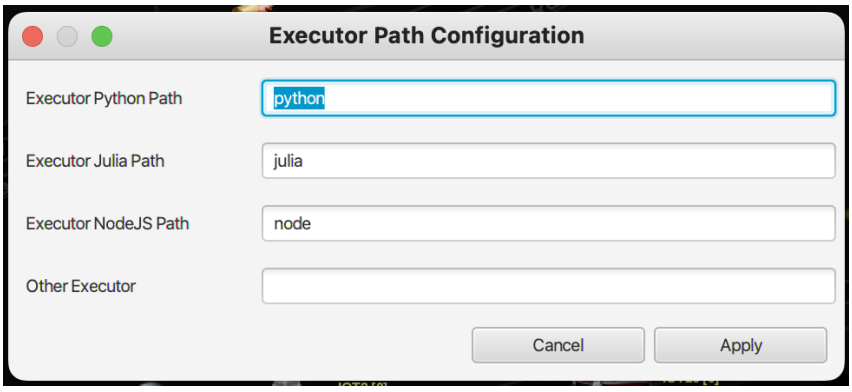

Figure 3. Executor Path configurations

The first path is the one used to run Python programs. It can be python, python3 or any other command that runs the Python programs. If you want to use the command of another version of

Python which is not the default one, you have to use the complete path of the python command of the other version. For example for macOS system, if the default version of the Python is 3.12 but you want to execute your programs using the version 3.10 then you write the path:

/Library/Frameworks/Python.framework/Versions/3.10/bin/python3

Verify that this version is installed before. The same case applies to the other commands. You can add commands for Python, Julia and NodeJS programs. The type of the sources are detected automatically by CupCarbon and it runs its corresponding command. If you are using another language (interpreted) different from the 3 languages considered by CupCarbon then use the 4the field (Other Executor) to define its execution command. In this case, if CupCarbon detects that the type is not known then it will use this path to execute it. In case where you are using executable files then leave the last path field empty.

Button 3: open the window of the programs

This button is used to open the window (cf. Figure 4) where you can add your programs.

Figure 4. Program Editor

This window is useful to quickly write your programs and it helps to get some samples to start with CupCarbon. You can find Python, Julia and JS samples such as:

- Hello World ! (Python)
- Alternate Hello and World! (Python)
- Display the info of an IoT sensor (Id, name, location, marked) in Python
- Blink (Python, Julia and NodeJS)
- Move (Python)
- MQTT Publisher (Python and Node JS)
- MQTT Subscriber (Python and NodeJS)

We recommend the use of an external code editor, such as Sublime Text or a similar tool. These editors offer several advantages, such as syntax highlighting, code autocompletion, code formatting, customization options, version control integration, multi-language support, strong community support, and optimized performance. The choice of code editor should depend on your personal preferences and project requirements. In this case, you must place your codes in the “scripts” folder within your project files as shown by the following Figure 5.

Figure 5. CupCarbon Project files

Button 4: run the simulation

Use this button to start your simulation.

Button 5: stop the simulation

Use this button to stop your simulation. Check the console to verify if there are some errors or not.

CupCarbon Project

To start with CupCarbon you need to create a new project. If you start with designing a new project without creating it before use the menu Project → new Project from the current. Refer to the previous User Guide for more details. It can be downloaded from the following link:

http://cupcarbon.com/cupcarbon_ug.html

After creating a new project, add one or many IoT nodes using the Button 1 presented above. It is possible to change the picture of each node by selecting it and pressing on the key j many times until you get the desired picture.

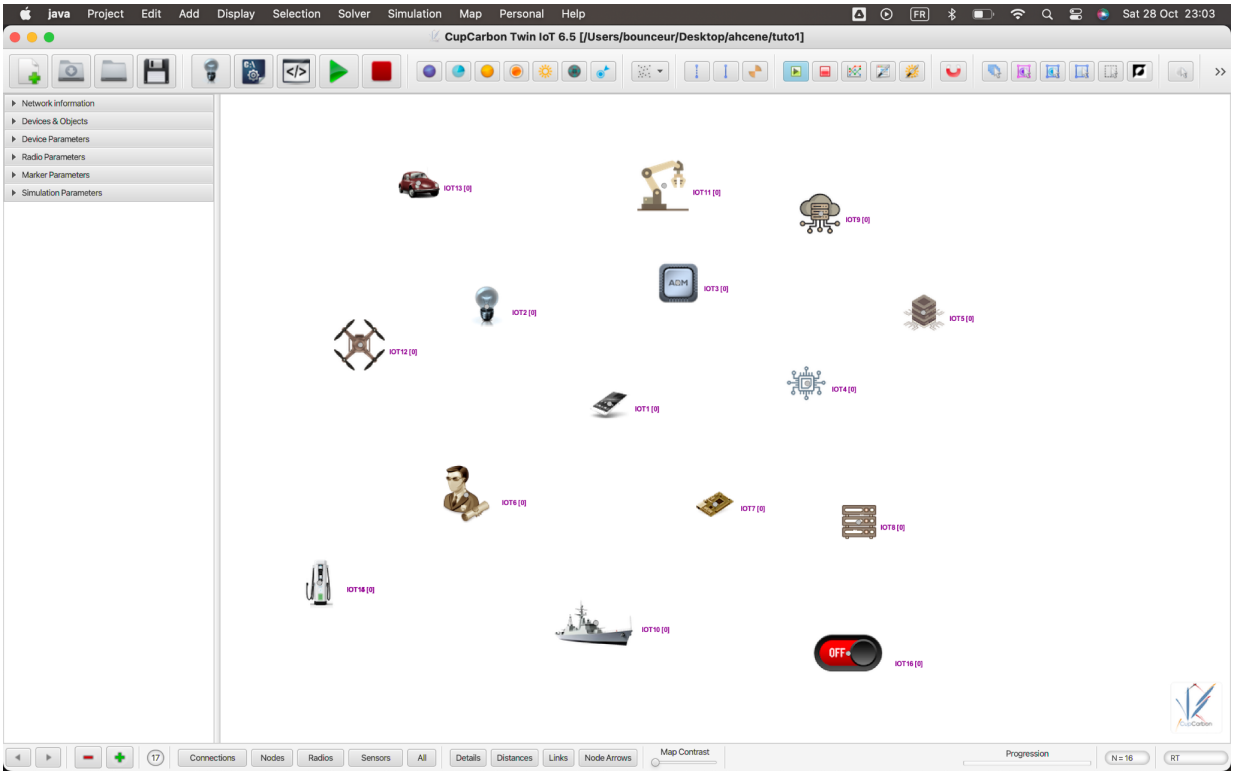

Figure 6. IoT nodes in CupCarbon

Hello World

This example shows how to write a code that allows to display a message or to do any action on the node executing that code.

To do that, from the Python code (or any other language) one must write the classical print command allowing to display strings in the console (terminal) in order to send a command to CupCarbon. For example, to display “Hello World!” one must use the print command of Python to send the print command of CupCarbon as shown in the following instruction:

```
print("print Hello World!", flush=True)
```

The first print is the Python command to print messages on the console. It will be sent to CupCarbon as well. CupCarbon will consider the received message “print Hello World!” as a command. If the command is known then it will execute it, otherwise, it will ignore it. In this instruction, the command is “print” and the message to print is “Hello World!” which means that the node of CupCarbon executing this code will display the message “Hello World!”. To send this message directly to CupCarbon directly one must use the command flush. In Python this command can be added directly in the print command or separately in another instruction using the library sys, like:

```
import sys
print("print Hello World!")
sys.stdout.flush()
```

To add a code that will be executed by a node, click on the Button 3 presented above to add a new code. The Program Editor (cf. Figure 4) will be opened.

It is possible either to write directly your code or using the proposed samples that can be added by clicking on the different buttons of this window. A sample of Hello World! code can be obtained by clicking on the first button. Once finished, add the name of the file in the field “File name” by specifying its extension. Because, it is possible to write code using any language you want. Finally, save the program using the button Save.

CupCarbon can consider Python, Julia and NodeJS by specifying their execution command in the window Executor Path Configuration. To open this window, click on the Button 2 presented above. You will then obtain the following window:

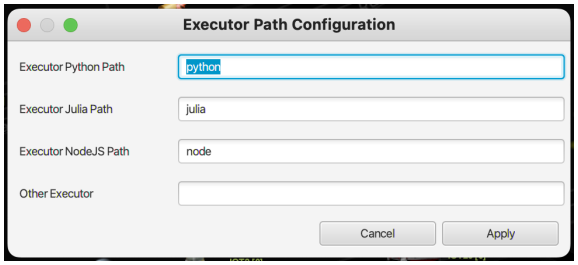

Figure 8. Executor Path Configuration

The first field represents the system command allowing to execute Python programs (it can be python3 or python). The second one is the one used to execute Julia programs and the third one is the one used to execute NodeJS programs. For any other language use its system execution command in the fourth field. If the execution of the program does not require any command such is the case for executable files, leave the fourth field empty.

Here are the codes of different languages to display Hello World! in CupCarbon

a. **Python:**

```
print("print Hello World!", flush=True)
```

b. **Julia:**

```
println("print Hello World!")
```

c. **NodeJS:**

```
console.log("print Hello World!")
```

d. **Java (.class):**

For Java, you have to use the .class generated after the javac command. The .class must be added in the folder scripts of the CupCarbon project.

```
public class Hello {
    public static void main(String [] arg) {
            System.out.println("print Hello World");
    }
}
```

To compile this Java file, use the command:

```
javac Hello.java
```

It will generate the file Blink.class in the same folder. This file must be added to the folder scripts of the CupCarbon project.

e. **C language (executable):**

Use the file generated after compilation. This obtained (exe) file must be added in the folder "scripts" of the CupCarbon project. The following program is a C program allowing you to display "Hello World!".

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    fflush(stdout);
    return 0;
}
```

To compile a c code in the file hw.c, one can use the following command:

gcc hw.c -o hw.exe

Blink (Mark/Unmark)

You can use the commands mark and unmark to blink any node. The code allowing to blink a node is given as follows:

a. Python

```
import time
while True:
      print("mark", flush=True)
      time.sleep(1)
      print("unmark", flush=True)
      time.sleep(1)
```

b. Julia

```
while true
      println("mark")
      sleep(1)
      println("unmark")
      sleep(1)
end
```

c. NodeJS

```
const {execSync} = require('child_process');
while (true) {
  console.log("mark")
  execSync('sleep 1')
  console.log("unmark")
  execSync('sleep 1')
}
```

d. Java

```
public class Blink {
    public static void main(String [] arg) throws Exception {
        while(true) {
            System.out.println("mark");
            Thread.sleep(1000);
            System.out.println("unmark");
            Thread.sleep(1000);
        }
    }
}
```

e. C language

```
#include <stdio.h>
#include <unistd.h>
int main() {
    while(1) {
        printf("mark\n");
        fflush(stdout);
        sleep(1);
        printf("unmark\n");
        fflush(stdout);
        sleep(1);
    } return 0;
}
```

Get Info

To use any parameter of an IoT node in the program use one of the following commands:

Command name	Action
getid	returns the id of the node
getname	returns the name of the node
getxy	returns the GPS location (longitude latitude) of the node
getx	returns the x location (longitude) of the node
getty	returns the y location (latitude) of the node
ismarked	returns “1” if the node is marked and “0” if not

Table 1. CupCarbon commands

The following program shows the Python instruction allowing to display the id of a node:

```
print("getid", flush=True)
id = input()
print("print My ID is: "+id, flush=True)
```

Move

The code Python to move an IoT to a given location is given as follows, where 50 and 40 are respectively the longitude where the node will be moved.

```
print("move 50 40", flush=True)
```

Use the sample "move" of CupCarbon for a complete example where an IoT node moves through a given route.

Control an IoT node from another IoT node (local)

CupCarbon offers the following two new features:

- Feature 1: the command **mark nid** that allows to mark the node with the id **nid** from the node executing this command
- Feature 2: it is possible to mark and unmark manually a node by clicking on it during the simulation.

These new features allow to mark/unmark a node from another node. As an example, we will create a project with two IoT nodes. One node will represent a switch and the other one will represent a lamp.

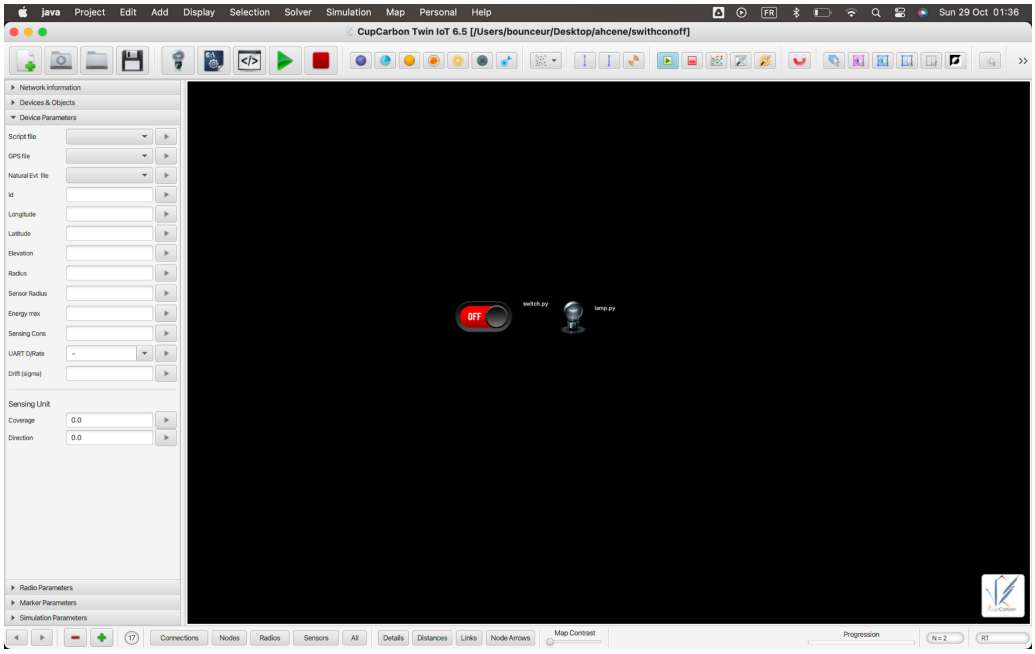

Figure 9. CupCarbon project with a switch and a lamp

During the simulation, the lamp will be switched on (marked) or switched off (unmarked) based on the state of the switch that will be marked/unmarked manually. Use the following steps to set up this project:

- Create a new project
- Add two IoT nodes, one with id=1 and one with id=2. To modify the id of an IoT node, select it, go to the Device Parameter panel on the left and modify the parameter Id, then click on the button in the right (with a small arrow) to assign the new id.

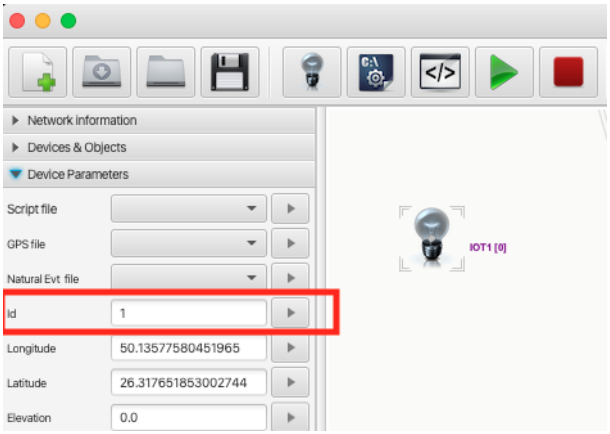

Figure 10. Parameters (id) of an IoT node.

- Change the picture of the IoT node 1 to obtain the switch (key j)
- Change the picture of the IoT node 2 to obtain the lamp (key j)
- Add the following Python code of the switch

```
import time
while True:
      print("ismarked", flush=True)
      id = input()
      if id=="1":
            print("mark 2", flush=True)
      else:
            print("unmark 2", flush=True)

      time.sleep(0.2)
```

- Add the following Python code of the lamp

```
import time
while True:
      print("ismarked", flush=True)
      x = input()
      print("MARKED = "+x, flush=True)
      time.sleep(0.2)
```

- Run the simulation (click on the switch to mark or to unmark it) ...

Control an IoT node from another IoT node (MQTT)

Add the following code of the **Publisher** (the switch). Change the topic **`cupcarbon/lamp`** by another one. You can change the broker (**broker.hivemq.com**) by another one as well.

```
from paho.mqtt import client as mqtt_client
import time
broker = "broker.hivemq.com"
port = 1883
topic = "cupcarbon/lamp"
print("getid", flush=True)
id = input()
client_id = "cupcarbon"+id
```

```
def connect_mqtt():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!", flush=True)
        else:
            print("Failed to connect, return code ", rc, flush=True)
    client = mqtt_client.Client(client_id)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client
def publish(client):
    while True:
        print("ismarked", flush=True)
        m = input()
        client.publish(topic, m)
        time.sleep(0.5)
def run():
    client = connect_mqtt()
    client.loop_start()
    publish(client)
    client.loop_stop()
if __name__ == '__main__':
    run()
```

Add the following code of the **Subscriber** (the lamp). Change the topic **`cupcarbon/lamp`** by another one which must be the same as the one of the Publisher. You can change the broker (**broker.hivemq.com**) by another one as well.

```
from paho.mqtt import client as mqtt_client
broker = "broker.hivemq.com"
port = 1883
topic = "cupcarbon/lamp"
print("getid", flush=True)
id = input()
client_id = "cupcarbon"+id
def connect_mqtt():
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!", flush=True)
```

```
        else:
            print("Failed to connect, return code", rc, flush=True)
    client = mqtt_client.Client(client_id)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client

def subscribe(client: mqtt_client):
    def on_message(client, userdata, msg):
        message = msg.payload.decode()
        if message=="1":
            print("print RECEIVED 1", flush=True)
            print("mark", flush=True)
        if message=="0":
            print("print RECEIVED 0", flush=True)
            print("unmark", flush=True)
    client.subscribe(topic)
    client.on_message = on_message
def run():
    client = connect_mqtt()
    subscribe(client)
    client.loop_forever()

if __name__ == '__main__':
    run()
```