Manticore

Clément Garnier Homère Faivre

July 10, 2014

Contents

Chapter 1

Sensors

The main purpose of this project is to allow the use of sensors distributed over a network. In the following, we propose a standardized procedure to describe a sensor and we also explain how it is incorporated within the framework.

Table of Contents *generated with DocToc*

1.1 Structure of `description.json`

Each sensor **must** be described by a JSON file called `description.json`. This file **must** be located at the root of the sensor working directory folder (each sensor has its own folder in the `sensors/` directory of the repository). The purpose of the file is to report the instructions that Manticore should follow to detect the presence of the sensor and to handle the request on it.

- `name`
  - *String*

  - **mandatory**
  - Name of the sensor and also the name of the folder containing the `description.json` file

- `systems`
  - *Object* containing *System* objects indexed by an alias
  - **mandatory** (at least one *Command* child)
  - Describes operating systems and architectures supported by the sensor. Each *System* is indexed by an alias that will be referenced by the `systems`property‘ of the *Command* objects

- `bootstrap`
  - *Array* of *Command* objects
  - **mandatory**
  - Describes the commands to be executing when Manticore starts in order to detect whether the sensor is present or not on the node

- `data`
  - *Array* of *Data* objects
  - **mandatory** (at least one *Data* child)
  - Describes the data and the related OSC format provided by a sensor

- `request`
  - *Object*
  - **mandatory**
  - Describes the Request procedure so that a node can request the sensor's data. This procedure can have several *modes* and contains 3 main steps namely *check* to check whether the sensor is still available, *generate* in the case that we need to generate a file and finally *execute* that will trigger the commands to send the data to the requester's endpoint.

1.1.1 System object

A system object is a data structure that describes the operating systems and architectures supported by a sensor. In the description file, that looks like the following :

```
"alias_of_system": {
  "platform": "platform_of_system",
  "arch": "architecture_of_system"
}
```

- The value of the `alias_of_system` can be anything that gives an understandable description to the system such as `linux`, `osx` or even `my_own_pc`.
  - Currently we use the following values `linux`, `pi`, `win` and `osx`
  - These aliases are custom-made and do not rely on Node.js API or terminology
- The `platform` property is **mandatory** and **must be equivalent** to a value returned by `require('os').platform()` in Node.js
- The `arch`property is optional and if set then it **must be equivalent** to a value returned by `require('os').arch()` in Node.js

  For more information, refer to the OS module in Node.js API

Pay attention to the difference between

```
"linux": {
    "platform": "linux"
  }
```

and

```
"pi": {
    "platform": "linux",
    "arch": "arm"
  }
```

The first one targets all Linux operating systems (regardless of the architecture) whereas the second one only targets those that run on ARM processors (but not specifically Raspberry Pi, the alias just implies it).

So if we have the following commands

```
{
  "cmd": "./command_for_unix_systems.sh",
  "systems": [
    "linux",
    "osx"
  ]
},
{
  "cmd": "./command_only_for_pi.sh",
  "systems": [
    "pi"
  ]
}
```

On the Rapsberry Pi, Manticore will match on 2 aliases `linux` (because of the platform) and `pi` (because of the platform and architecture) and therefore execute both commands.

Here, we use the alias `pi` for because we use devices called Raspberry Pi. Nonetheless the alias `linux-arm` could also have been used because its meaning is more closely related to the platform and architecture description.

The important thing is to choose an alias that fits best to what you want to achieve and to be consistent in the way of describing a specific platform or architecture within one `description.json` file. Indeed, these system aliases are going to be used as a reference in the `systems` property of the Command object (described in the next section).

1.1.2 Command object

The Command object is a data structure representing a command that must be executed to perform any action.

1.1.2.1 Command structure

```
{
  "path": "subfolder/scripts",
  "cmd": "./my_script.sh",
  "parameters": [
```

```
    "--addr"
    "$ADDRESS",
    "--port",
    "$PORT",
  ],
  "systems": [
    "osx",
    "linux"
  ],
  "sudo": true
}
```

- The `path` property is the place where the subshell will be executed. This property is optional if the command is in the environment variable `PATH` or if the command/program is in the same directory as the `description.json` file.
- The `cmd` property is the command that will be executed. This property is obviously **mandatory**.
- The `parameters`is an *Array* of arguments that will be passed to the command specified in the `cmd` property
  - Parameters starting with a `$` (dollar sign) are variables. This means that they are going to be parsed and their value replaced by Manticore. For instance, the `$ADDRESS` and `$PORT` in the above example are variables.
- The `systems` property is an *Array* of aliases to System objects that specifies on which systems you can apply the command. This property is **mandatory**.
- The `sudo` property is optional and when sets to `true` implies that the command must be executed with superuser rights.

Thus, considering that we are in the sensor working directory, the result of the above example is equivalent to execute

```
$ cd subfolder/scripts
$ sudo ./myscript.sh --addr 127.0.0.1 --port 42424
```

Note that the variables `$ADDRESS` and `$PORT` have been respectively replaced by values `127.0.0.1` and `42424`.

1.1.2.2 Note about the execution of the command

You may notice that for a shell script the `cmd` often starts with `./myscript.sh`. This is because Manticore will create a subshell in the sensors folder and then execute the content `cmd` from there.

If you forget the './', then it will try to execute `myscript.sh` and therefore look for the environment variable `PATH` for it. Of course, the sensor folder is not in the `PATH` and you will get an error like `myscript.sh: command not found` or `myscript.sh: No such file or directory`.

1.1.2.3 Command exit status

For each sensor, Manticore is responsible for parsing the `description.json` file and thus executing the commands described in the Command object.

To achieve this goal, we are using the Child Processes module in Node.js API. This means that Manticore will execute the command in a child process and will monitor its life cycle.

Indeed, when the child process exits, Manticore will inspect the command return value (or exit code). If it is a success then it will jump to the next command, otherwise an error/exception must be thrown.

By convention, a **success** is denoted by the `0` **value**. As a consequence, any other value that defers from `0` will be considered as a failure.

> In UNIX-like shells, the exit code of the precedent command can be displayed with `echo $?`

1.1.3 Data description object

The Data object gives a description of the data provided by a sensor.

```
{
  "description": "X position of the mouse",
  "osc_format": "/mouse/x f"
}
```

The object has a simple structure with 2 properties :

- The `description` property provides a simple text description of the data
- The `osc_format` property shows the syntax of the OSC address pattern and type tag.
  - In the above example, `/mouse/x` is the OSC address and `f` the type tag for floating-point numbers

> For more information about OSC, refer to the specification

Remember that the OSC format must be in accordance with the program that is responsible to forge the OSC packets and to send (i.e. the program triggered by the *execute* step in Request procedure).

1.1.4 Bootstrap procedure

The Bootstrap procedure corresponds to the commands that must be executed to detect the sensor on a node.

The structure is quite simple and is a simple array of Command objects for all supported systems

```
"bootstrap": [
  {
    "cmd": "./detectionScript-osx.sh",
    "systems": [
      "osx"
    ]
  },
  {
    "cmd": "./detectionScript-linux.sh",
    "systems": [
      "linux"
    ]
  }
]
```

When Manticore starts up, it will execute these commands to detect the presence of the sensor on the node. If it is a success then, the sensor's presence will be published across the network.

1.1.5 Request procedure

The Request procedure corresponds to the ability to send the sensor's data in OSC format to another node that have requested it.

```
"request": {
    "default": {
      "options": [
        "$ADDRESS",
        "$PORT",
        "$GENERATED_PATCH"
      ],
      "check": [
        // some Command objects
      ],
      "generate": [
        // some Command objects
      ],
      "execute": [
        // some Command objects
      ],
    }
}
```

The request procedure is already implemented into Manticore, so it means that you can create new sensors and therefore no ad-hoc development should be required to make the request of the sensor's data available to other nodes. Everything is defined here in the description file.

1.1.5.1 Mode

As you can notice on the above excerpt of `request`, it has a property called `default`. We will refer in the following as the *mode* of the Request procedure. This will help us to give some granularity in the request procedure.
Usually, there will only be one `mode` called `default`. Therefore the `default` mode is **mandatory**, others are optional.

How is this *mode* hierarchy be useful ? Let's consider a situation where we might want a slightly different type of the Request procedure. It is easy to implement it by just adding a new property after `default`.
As previously defined, the `default` procedure objective is *to send the sensor's data in OSC format to some other node*. Here, we can think of other *modes* that can be *send the sensor's data with another type of format* or *to send the data over TCP (instead of UDP)*.

1.1.5.2 Limitation

The `limitation` property of the `default` mode of the Request procedure sets the maximum of concurrent clients that can request the data of a sensor.

This is to take in account the limitation of some sensors or drivers

> WARNING, this is a new specification that is not yet implemented in Manticore

1.1.5.3 Options

Each mode can have some options. These options corresponds to variables that will be set by Manticore at runtime.

The convention used here is to write them starting with a `$` (dollar sign).

The Request procedure have 3 options (in `default` mode) :

- `$ADDRESS` corresponds to the IP address of the endpoint
- `$PORT` corresponds to the port of the endpoint
- `$GENERATED_PATCH` corresponds to the name of a generated file that can be needed before triggering the execution

The name of the option is just a way to remember and describe it. If your need to generate a script and not a patch, then you can call the third options `$MYSCRIPT` instead of `$GENERATED_PATCH` and use this name in your `description.json`. This will not affect the parsing of the file.

However the order is very important because Manticore will process these variables in the same order that are they described above (first the address, then the port and finally the generated file name). This means that for each *mode* of a certain procedure, the order and the number of the options must be consistent across all sensors' description files.

1.1.5.4 Steps: Check > Generate > Execute

The `default` mode of the Request procedure is divided into 3 steps that must be run one after another

1. `check` Check whether the sensor is still available
2. `generate` Generate an executable or a script at runtime (optional step)
3. `execute` Execute an existing or previously generated executable/script that will send the data to the endpoint that requests the resource

Each of the above mentioned steps are just arrays of Command objects that are going to be executed in order in regards of the `systems` property.
For the two first steps, Manticore will monitor the exit code checking whether the `check` and `generate` steps were a success. For the `execute` step, Manticore will run the command indefinitely until he gets a request for releasing the resource.

> For more details, you can look at the implementation in the *Sensor* constructor in `sensor.js` and especially the method `request()` and the functions `executeCommand()`, `parseAndExecute()`, `parseExecuteAndDie()`.

1.2 How is this description file used by Manticore ?

At the startup of Manticore, the program will try to detect the presence of sensors on the node. To achieve this goal, Manticore will browse the content of each folder in `sensors/`. Each of these folders are the working directories of a specific sensor and thus must contain a `description.js` file.

This description file – which content is described in the previous section – will be parsed by Manticore (for those interested in the implementation, you can refer to the method `Core.prototype.detectSensors` in `manticore.js`).

1. The first element parsed is the `systems`. According to the node's platform and architecture, Manticore will determine which system aliases that the node is entitled.
2. Then Manticore will try detect the sensor on the current node. To do so, it parses the `bootstrap` element and browses the Command. For each Command, Manticore checks its `systems` property for matches with the system aliases. If it success, then the `cmd` is executed with `parameters`. In terms of implementation, this is done in the *Sensor* constructor (see `sensor.js` file), if the `bootstrap` fails (either because the sensor is not entitled to the node's system or because), then constructor should not return a new *Sensor* object and fail. If it is a success, the new *Sensor* object is created and the *Core* singleton gets aware of it in its own `sensors` property. Thereafter all the detected sensors are published across the network.
3. When the *Sensor* object is created. Not only it detects it and sets up its properties(identifier, name and associated data) but also automatically implements a method `request()` matching the instructions of the Request procedure in the description file.
    - The prototype of this function is simple `request(mode, array_of_options)`.
    - The `mode` will try match the one in the description file (if not set, automatically use `default`).
    - The `array_of_options` are parameters set up by Manticore (e.g. after a request by another node) and therefore matched to the `options` in the description file.
4. Then Manticore will execute each step of the procedure one after another. The next step cannot be triggered if the previous step has not finished successfully. If one step fails then the procedure cannot come to a successful conclusion and an error is triggered.

    In the implementation, in order to avoid a callback hell and to give some modularity in the code, we are using the async module to execute the steps one after another

1.3 Tutorial: Adding a sensor

1.3.1 Setting up the workspace

As stated above, the repository contains a `sensors` folder which contains all the sensors.

```
$ cd $REPO_ROOT/sensors
$ mkdir my_new_sensor
```

You should now have the following tree view

```
$REPO_ROOT
  |____sensors
  | |____inertial
  | | |____description.json
  | | |____ ...
  | |____mouse
  | | |____description.json
  | | |____ ...
  | |____my_new_sensor
  | | |____description.json
```

1.3.2 Write the description file

1. Create a `description.json` file with an empty object

```
{
}
```

2. Add the `name` property and sets its value to the name of sensor (which must also be the name of the folder containing the description file)

```
{
  "name": "my_new_sensor"
}
```

3. Add the `systems` property that will contain objects describing the platform and architecture supported by the sensor. Here we target Linux, Mac OS X and Windows operating (regardless of system versions and architecture)

```
{
  "name": "my_new_sensor",
  "systems": {
    "linux": {
      "platform": "linux"
    },
    "osx": {
      "platform": "darwin"
    },
    "win": {
      "platform": "win32"
    }
  }
}
```

4. Add the `data` description object that will describe the type of data provided by the sensor

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [
    {
      "description": "some data float value",
      "osc_format": "/sensor/data f"
    },
    {
      "description": "some button on/off",
      "osc_format": "/sensor/button i"
    }
  ]
}
```

5. Describe the `bootstrap` procedure (i.e. the command used by Manticore at startup to detect the sensor). As we have stated above that the sensor is available on Linux, Mac OS X and Windows, we need to describe the commands accordingly.

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [ ... ],
  "bootstrap": [
```

```
    {
      "cmd": "./checkMySensor-unixlike.sh",
      "systems": [
        "osx",
        "linux"
      ]
    },
    {
      "cmd": "checkMySensor-windows.bat",
      "systems": [
        "win"
      ]
    }
  ]
}
```

6. Describe the `request` procedure and add a `default` mode (it's mandatory, at least one mode)

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [ ... ],
  "bootstrap": [ ... ],
  "request": {
    "default": {
    }
  }
}
```

7. The `request` procedure is standardized with 3 options: the address of the endpoint (`$ADDRESS`), the port of the endpoint (`$PORT`) and a name for the generated script/executable if needed (`$GENERATED_PATCH`).

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [ ... ],
  "bootstrap": [ ... ],
  "request": {
    "default": {
      "options": [
        "$ADDRESS",
        "$PORT",
        "$GENERATED_PATCH"
      ],
    }
  }
}
```

8. For the `check` step, we will use the same scripts previously used in the `bootstrap` procedure

```
{
  "name": "my_new_sensor",
  "systems": { ... },
```

```
  "data": [ ... ],
  "bootstrap": [ ... ],
  "request": {
    "default": {
      "options": [ ... ],
      "check": [
        {
          "cmd": "./checkMySensor-unixlike.sh",
          "systems": [
            "osx",
            "linux"
          ]
        },
        {
          "cmd": "checkMySensor-windows.bat",
          "systems": [
            "win"
          ]
        }
      ]
    }
  }
}
```

9. For the `generate` step, we will assume that no generation is required here (for instance, the mouse sensor need a Pure Data patch whereas the inertial sensor does not need anything)

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [ ... ],
  "bootstrap": [ ... ],
  "request": {
    "default": {
      "options": [ ... ],
      "check": [ ... ],
      "generate": [
        // empty !
      ]
    }
  }
}
```

10. For the `execute` step, we will use some parameters that are variables.

```
{
  "name": "my_new_sensor",
  "systems": { ... },
  "data": [ ... ],
  "bootstrap": [ ... ],
  "request": {
    "default": {
      "options": [ ... ],
      "check": [ ... ],
      "generate": [],
      "execute": [
```

```
            {
              "cmd": "./sendData-unixlike.sh",
              "parameters": [
                "$ADDRESS",
                "$PORT"
              ],
              "systems": [
                "osx",
                "linux"
              ]
            },
            {
              "cmd": "sendData-windows.bat",
              "parameters": [
                "$ADDRESS",
                "$PORT"
              ],
              "systems": [
                "win"
              ]
            }
          ]
        }
      }
    }
```

1.3.3 Write your scripts/programs

According to the written `description.json` file, now you should write your scripts or dedicated programs and drivers.

For the `bootstrap` procedure and the `check` step of the `request` procedure

- `checkMySensor-unixlike.sh`
- `checkMySensor-windows.bat`

Both of these scripts must return an exit code `0` if the test is a success.

For the `execute` step of the `request` procedure

- `sendData-unixlike.sh` that takes 2 arguments (address and port)
- `sendData-windows.bat` that takes 2 arguments (address and port)

Both of these scripts must trigger a program that will send OSC data to the endpoint

1.4 Custom procedure

If you want to create a custom procedure for your sensor. We can look at the Request procedure or the skeleton below.

```
"procedure_name": {
    "default": {
      "options": [
```

```
    // some options
      ],
      "step_1": [
        // some Command objects
      ],
      "step_2": [
        // some Command objects
      ],
    },
    "custom_mode": {
      "options": [
    // some options
      ],
      "step_1": [
        // some Command objects
      ],
      "step_2": [
        // some Command objects
      ],
    },
}
```

It is important to understand that – contrary to the Request procedure – any new procedure implies that some ad-hoc development must be done in Manticore in order to handle it.
To implement, you should edit the constructor of the *Sensor* object (`sensor.js`) and implement a new method called `procedure_name()`. Doing so, you will then be able to call it in a standard way `my_sensor.procedure_name(mode, [args])` with `mode` being either `default` or `custom_mode` and the array of arguments matching to the `options` property of the considered mode.

1.5 Further works

- Find a way to create some JSON Schema and to validate the JSON description files, maybe see http://json-schema.org/
- Automatic generation of methods regarding the custom procedure, we could parse all other procedure and get the method associated, that will use the `async` module to execute each step. The code will be then generated at runtime
- We can think to develop a workaround for the limitation of some sensors driver in the case of multiples concurrent client requests. To do so, the data would always be sent to Manticore and Manticore would be responsible to duplicate the OSC data and to send them to multiple client at the same time. However, this means that we need to change the way Manticore handles and respond to requests

Chapter 2

Manticore

In the following, the words *core* and *manticore* with/without a capital letter are used indistinctly and refer to the same program.

Table of Contents *generated with DocToc*

- Avahi warning on Linux
- Auto-detection on multiple interfaces

2.1 Getting started

To start Manticore, you can either use node

```
$ node app.js
```

or npm

```
$ npm start
```

on Windows, use `node.exe`

```
> node.exe app.js
```

2.2 Guide

2.2.1 Introduction

// TODO : simply detail use case and present the associated objectives

2.2.2 Objectives

2.2.2.1 What we want to achieve ?

- **Discovery**
  How do we learn about other nodes on the network ?
- **Presence**
  How do we track when others nodes come and go ?
- **Connectivity**
  How do we connect one node to another ?
- **Group messaging** (a.k.a. multicast)
  How do we send a message from one node to a group of other nodes ?
- **Point-to-point messaging**
  How do we send a message from one node to another ?
- **External communication**
  How does a node communicate with process that are not *network nodes* ?
- **Content distribution**
  How do we provide the requested resource data ? How do we send the data ?

2.2.2.2 How does it work ?

- At startup, a node **advertise** its **presence** by broadcasting a `_node._tcp` service on **Zeroconf**. Doing so, other nodes can simply browse this service to dynamically discover its presence.
- We assume that all the nodes are on the **same network segment** (i.e in the same subnet). Thus we can infer that they can directly address other nodes using the recipient IP address.

- Each node provides different **communication channels** relying on a specific pattern : **uni-** or **bidirectional**, **synchronous** or **asynchronous**.
- The **group messaging** is achieved by a communication channel called *Information Channel (InCh)* implementing the publisher/subscriber network pattern. Hence, each node has 2 sockets :
  - the publisher socket solely used to send information to all its subscribers.
  - the subscriber socket solely used to receive information from all other nodes.
- The **point-to-point messaging** is achieved by another communication channel called *Main Channel (MaCh)* implementing the request/reply network pattern. According to the situation, this request/reply can either be synchronous (an immediate reply is requested and thus will block the execution) or asynchronous (we expect a response but not urgently).
- To communicate with **external processes**, a HTTP server is embedded in each node and can provide a **web API**, accepting GET request and serving JSON file or plain text status.
- The **distribution of the resource** data is made using **OSC packets** on UDP datagrams. The main use case is a client that desire a resource provided on a specific node of the network. To achieve a proper delivery of the dynamic OSC data stream to this client, we use the information collected on both InCh and MaCh and execute the following procedure :
  1. A client issue a request on a local core to know the network status and resources available
  2. The server sends back a list of network nodes with its capabilities (this list is known by the core through 1. Browsing of `_node._tcp` service and 2. Listening on any published event on InCh)
  3. The client issue a request on a resource, binds an UDP reception socket and wait for the core's response
  4. The local core knows which node to ask and issue in turn a synchronous request using MaCh. This is a synchronous request, meaning that it will waits for a reply.
  5. The recipient core handles this request, check the availability of the requested resource, if so it sends a positive reply to the requester core. At the same time, it may generate a file and execute a program/driver that will send the data to the requester client.
  6. The local core receives the response, process it and can now answer the client.
  7. A OSC stream over UDP must now be flowing between from the node with the requested to resource to the client

Note: Both *MaCh* and *InCh* need to be reliable communication channels and thus use TCP as transport protocol.
Note 2: These channels are solely used to inter-core communication. That is to say that any external communication with a core must use the built-in HTTP server.

2.2.2.3 Note about the use of ZeroMQ Sockets

- PUB to publish data
- SUB to subscribe to a PUB socket
- REQ to issue a synchronous request
- REP to issue a synchronous reply (not used actually, see ROUTER)
- ROUTER to issue asynchronous replies
- DEALER to issue asynchronous requests

We can think of REQ and DEALER soscket sas "clients" and ROUTER sockets as "servers". That is why we bind the ROUTER sockets and connect REQ and DEALER sockets to them.

> //TODO: write about 0MQ sockets and pattern Node: see bind/connect mechanism in UNIX sockets

2.2.2.4 Note about the use of mDNS

We use the mdns module for Node.js.

See the documentation : http://agnat.github.io/node_mdns/user_guide.html

2.2.2.5 Use cases

- `PUB[1] --remote--> SUB[N] | ROUTER[N] --output--> DEALER[1] --ack--> ROUTER[N]` (mixing InCh + MaCh and used for remote command execution)
- `REQ[1] --request--> ROUTER[2] --ack|noack--> REQ[1]` (using MaCh to request a resource)

// TODO : detail the procedure

2.2.3 Interactive commands

The following commands are available

- `debug` show core.nodes
- `eval [js]` use eval() javascript function to imitate REPL mechanism
- `log [js]` same as eval but will also show the result on stdout using console.log()
- `send [msg]` send a the string `msg` to publish socket
- `remote [cmd]` ask remote execution of `cmd` command
- `emit [event]` is equivalent to core.emit('event'), used for debug purpose only
- `exec [cmd]` execute a command in the shell (no need to quit the program or to open a new ssh session)

- `exit` gracefully close sockets and exit (equivalent to Ctrl+C)
- `request`

2.2.4 Events

- `ready` is triggered when **initialization finishes**
- `inch` is triggered when the **subscriber** socket **receives** some data (meaning *I've just received some data on the information channel* or *Another node just published some information*)
- `mach` is triggered when a **request** is **received** on MaCh (meaning *I've just received a request*)
- `reply` is triggered when a **response** to **previous request** is received
- `died` is triggered when we discover that a **node disapears** from the network
- `test` (for testing purpose only)

2.2.5 Core commands

These are the methods used by the Core singleton to interact with its state and its communication channels.

- `init()` will start mDNS advertising and browsing of `_node.tcp` service, bind sockets
- `publish()` will trigger a `inch` event on all subscribers
- `send()` will trigger a `mach` event on the recipient
- `syncSend()` will also trigger a `mach`
- `reply()`

- `close()`

  // TODO : add commented examples

2.2.6 Source files

- `app.js` is the main entry point of the program
- `manticore.js` is the module containing the core singleton
- `node.js` is the Node class
- `interactive.js` contains code for interactive commands in the shell
- `trigger.js` contains code for generating and executing processes

2.2.7 Data structures

- Message
- Node
- Sensor
- Record

2.2.7.1 Record

The Record class is used to keep track of requests, states and history of the core activity and communication.

The implementation of the Record class is located in `record.js` and used either in the main application `app.js` and the Core `manticore.js`.

We distinguish 2 types of records :

- `active_resource` : these records track every resource that have been requested by another core node and are aware of :
  - the time of the reception of the `request` command on MaCh
  - the core node id which requested the resource (so he can identify it in `core.nodes[]`)
  - the IP address and port of the endpoint where to send the sensor data (namely OSC packets)
  - ulimately if the core node that requested the resource is on the same machine as the client the IP address of the endpoint should match the IP of the node id (but this is not mandatory)
- `client_request` : these records track every request issued by a local client and is aware of :
  - the time of the reception of the method GET `/request/[id]?[port]` on the built in HTTP server
  - the IP address of the client (if local client, then 127.0.0.1)
  - the port that the client requested that we sent data on
  - the IP of the destination node that can provide the data (this is used as a commodity for the release resource procedure, so we do not need to look up the core.nodes[] and associated sensors[] to find the IP address)

These records works in pairs and at any time if a `client_request` is a valid record in one node (i.e. the client), you could fine an equivalent `active_resource` on another node of the network (i.e. the server). This `active_resource` is matching to the previously initiated `client_request`.

2.2.8 Inter-core messaging

2.2.8.1 Message structure

The message exchanged on the communication channels (InCh and MaCh) are JSON files and have the following structure :

```
{
  "header": {
    "src": "4626ca80-f211-11e3-8ad9-4df458080716",
    "name": "clementpi",
    "ip": "192.168.1.171",
    "type": "raw"
  },
  "payload": {}
}
```

It is simply a Javascript object with 2 main parts :

- `header` contains the type of message and some information about the sender (uuid, hostname and IP address)
- `payload` can be any Javascript primitive data types (i.e. `String`, `Boolean` or `Number`), composite data types (i.e. `Object` or `Array`) or special data types (i.e. `null` or `undefined`). In the case of composite types, its structure will be related to the type of message specified in the header.

  // TODO : need to write about ZeroMQ Frame and envelope

2.2.8.2 Message command and associated payload

- `raw`
- `request`
- `release`
- `ack`

2.2.9 External messaging

Inspired by REST API Using GET HTTP request

2.2.10 Reading the log

// TODO write about conventions used in logging

The logging have the following structure

```
<symbol>[<subject>] <message>
```

Like this (starting procedure)

```
+[CORE] Core starting on macbook-cgarnier.local
+[CORE] Core id 506243e0-02b9-11e4-87f0-8dea239e7eaf
+[HTTP] Listening on 3000
+[UDP]  UDP socket listening on 0.0.0.0:42424
+[PUB]  Publisher socket listening on 32323
+[CORE] Advertising _node._tcp on 32323
+[mDNS] Start browsing for _node._tcp services
+[MACH] Socket listening on 45454
```

The `<symbol>`can be

- `+` used for any relevant information
- `>` used for incoming message
- `!` used for errors
- `-` used for the disapperance of a node

The `<subject>` can be

- `CORE`
- `HTTP` for any connection or event related to the Web user interface
- `UDP` for the creation and the reception on the built-in UDP socket
- `mDNS` for anything related to the mdns module (browsing and advertising)
- `MACH` for any message on the main channel (MaCh)
- `INCH` for any message on the information channel (InCh)
- `SYNC` for any synchronous request
- `ASYN` for any asynchronous request
- `RELR` for the Release Resource procedure
- `REQR` for the Request Resource procedure
- `REQ` for anything related to a request (either synchronous or asynchronous)
- `REP` for anything related to a reply
- `PUB` for anything related to the publisher socket (used by InCh)
- `SUB` for anything relted to the subscriber socket (used by InCh)

2.2.11 HTTP Web user interface

2.2.11.1 Jade

// TODO Jade templating engine // Add a screenshot

2.3 Installation

2.3.1 Prerequisites

Manticore is based on Node.js and exploits ZeroMQ and Zeroconf/Bonjour.

2.3.1.1 Prerequisites on Mac OS X

Clone this repository

```
$ git clone https://github.com/garnierclement/pfe
```

Install the package manager Homebrew

```
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
$ brew update
```

Install Node.js v0.10.28

```
$ brew install nodejs
```

Install ZeroMQ 4.0.4

```
$ brew install zeromq
```

Note : pkg-config may be required too (`brew install pkg-config`)

2.3.1.2 Prerequisites on Raspbian (Raspberry Pi)

Clone this repository

```
$ git clone https://github.com/garnierclement/pfe
```

Update package manager

```
$ sudo apt-get update
```

Install Avahi daemon and its tools

```
$ sudo apt-get install avahi-daemon libnss-mdns
$ sudo apt-get install avahi-utils
```

Install Pure Data

```
$ sudo apt-get install puredata
```

Compile and install Node.js v0.10.28 from source tarball (**on the Raspberry Pi, it takes around 2 hours !**)

```
$ wget http://nodejs.org/dist/v0.10.28/node-v0.10.28.tar.gz
$ tar xvzf node-v0.10.28.tar.gz
$ cd node-v0.10.28
$ ./configure
$ make
$ sudo make install
$ sudo ldconfig
$ rm zeromq-node-v0.10.28.tar.gz
$ rm -rf node-v0.10.28
```

Alternatively, if you have already compiled it once (i.e. `make`) on a Raspberry Pi, you can directly do the following

```
$ cd node-v0.10.28
$ ln -fs out/Release/node node
$ sudo /usr/bin/python tools/install.py install
$ node -v          # must display v.0.10.28
$ npm -v         # must display v1.4.9
```

Compile and install ZeroMQ v4.0.4 from source tarball (it also requires libtool, autoconf and automake, but they are provided by Rapsbian)

```
$ sudo apt-get install uuid-dev
$ wget http://download.zeromq.org/zeromq-4.0.4.tar.gz
$ tar xvzf zeromq-4.0.4.tar.gz
$ cd zeromq-4.0.4
$ ./configure --with-pgm
$ make
$ sudo make install
$ sudo ldconfig
$ rm zeromq-4.0.4.tar.gz
$ rm -rf zeromq-4.0.4
```

2.3.1.3 Prerequisites on Windows

This is experimental and only tested on Windows 7 x64.

1. You need to dowload and install
   - Node.js v0.10.x
   - Bonjour SDK for Windows
     To download the SDK, you need to have an Apple Developer ID (it's free)
     Bonjour SDK is required for the dns_sd.h header file used by the mdns module for Node.js
     At the time of writing version 2.0.4 is the latest available
   - Python 2.7
     Python is required when installing the zmq module for Node.js
     During the installation, check the box *add python.exe to PATH*
   - ZeroMQ 4.0.4
     During the installation, choose the full installation (with source code and compiled librairies)
     Direct link for Windows x64
     Direct link for Windows x86
   - Git or GitHub for Windows (optional)
2. Reboot your system

2.3.2 Node.js module dependencies

Install Node.js module dependencies (mdns and zmq) with
(see `package.json` for more information about versions)

```
$ npm install
```

If you encounter any issue with the automatic npm installation, you can manually install them one by one.

```
$ npm install mdns
$ npm install zmq
$ npm install express
$ npm install jade
$ npm install uuid
$ npm install underscore
$ npm install async
```

Note: depending on the operating system zmq and mdns have other requirements (see the Prerequisites for your system above).

2.3.3 Let's go

Now that everything is set up, you can move up to Getting started

2.4 Known issues

2.4.1 Avahi warning on Linux

On Linux distributions, we can see the following warning when executing the program

```
*** WARNING *** The program 'node' uses the Apple Bonjour compatibility layer of Avahi.
*** WARNING *** Please fix your application to use the native API of Avahi!
*** WARNING *** For more information see <http://0pointer.de/avahi-compat?s=libdns_sd&e=node>
*** WARNING *** The program 'node' called 'DNSServiceRegister()' which is not supported (or only s
*** WARNING *** Please fix your application to use the native API of Avahi!
*** WARNING *** For more information see <http://0pointer.de/avahi-compat?s=libdns_sd&e=node&f=DNS
```

This must not be harmful and the warning can be hidden with the following environment variable

```
$ export AVAHI_COMPAT_NOWARN=1
```

For further investigation, enquire http://0pointer.de/avahi-compat?s=libdns_sd&e=node

2.4.2 Auto-detection on multiple interfaces

Because a node advertises a *node*.tcp service and at the same time browses/discovers any node advertising the same *node*.tcp service, it is ultimately going to see multiple instance of itself.

Currently we only care about the first one it sees (but maybe it is not the one we want) and for the others, the log will show something like

```
+[mDNS] Service up: macbook-cgarnier at 192.168.190.1 (vmnet1)
![CORE] Node id 506243e0-02b9-11e4-87f0-8dea239e7eaf is already present
```

So when a computer have multiples network interfaces, it is going to see itself multiple time.

Note that this behavior is also shown with virtual interfaces such as `vmnet1` or `vmnet8` used by VMware.