Rapport final
-
Grape Cluster

Author :
Cyril Bos
Paul Breton
Benjamin Dos Santos
Nicolas Hannoyer
Marion Lafon
Jason Pindat
Nicolas Vidal

Encadrant :
David Renault

Client :
Floréal Morandat

7 avril 2017

Table des matières

Introduction

Dans notre société où les problématiques liées aux réseaux sont de plus en plus sur le devant de la scène, il peut devenir enrichissant d'effectuer, et de simuler des situations supposées "réelles" dans des environnements "clos" et contrôlés. C'est dans ce but que le Grape Cluster a été pensé. Né dans le cadre d'un projet d'école, il permet de réaliser des "TPs" et des expériences réseaux en utilisant des machines physiques réelles plutôt que des machines virtuelles. Il consiste en une carte électronique permettant de connecter six raspberries pi, qui seront désignés comme esclaves, et contrôlés par un septième qui sera le maître. De plus ces cartes peuvent en théorie être chaînées pour former des clusters allant jusqu'à 96 raspberries. Ainsi nous pouvons obtenir certains cas qui ne peuvent être émulés, par exemple une machine qui tombe subitement en panne. On obtiendra ainsi des erreurs habituellement invisibles lors de l'utilisation de machines virtuelles. Ce cluster est également nettement moins coûteux qu'un parc de machines réelles.

Le but de ce projet a ainsi été de réaliser une interface permettant de configurer et surtout d'interagir avec un Grape Cluster, dont toute la partie électronique avait déjà été réalisée. Le cluster est composé de bornes (Boîtes sur le schéma 1) interconnectées par un bus i2c (**1**). Il est dirigé par un seul raspberry pi dit *maître* (**2**) contenu dans une des bornes. La borne contenant le *maître* contient donc 6 autres raspberries pi dits *esclaves* (**3**). Les autres bornes contiennent seulement 6 *esclaves*. Chaque raspberry pi présente 3 ports i2c dont un est relié au bus i2c de sa borne. Les bornes communiquent ainsi grâce à leurs interconnexions i2c. Chaque borne contient également des sondes (**4**) permettant de récupérer la température et l'utilisation électrique de chacune de ses raspberries ou de la totalité d'une borne. Un autre composant électronique permet de couper l'alimentation d'un raspberry souhaité (**5**). Une communication avec l'extérieur est également possible grâce à des switch réseau (**6**). Cette structure peut ainsi se représenter comme sur le schéma 1.

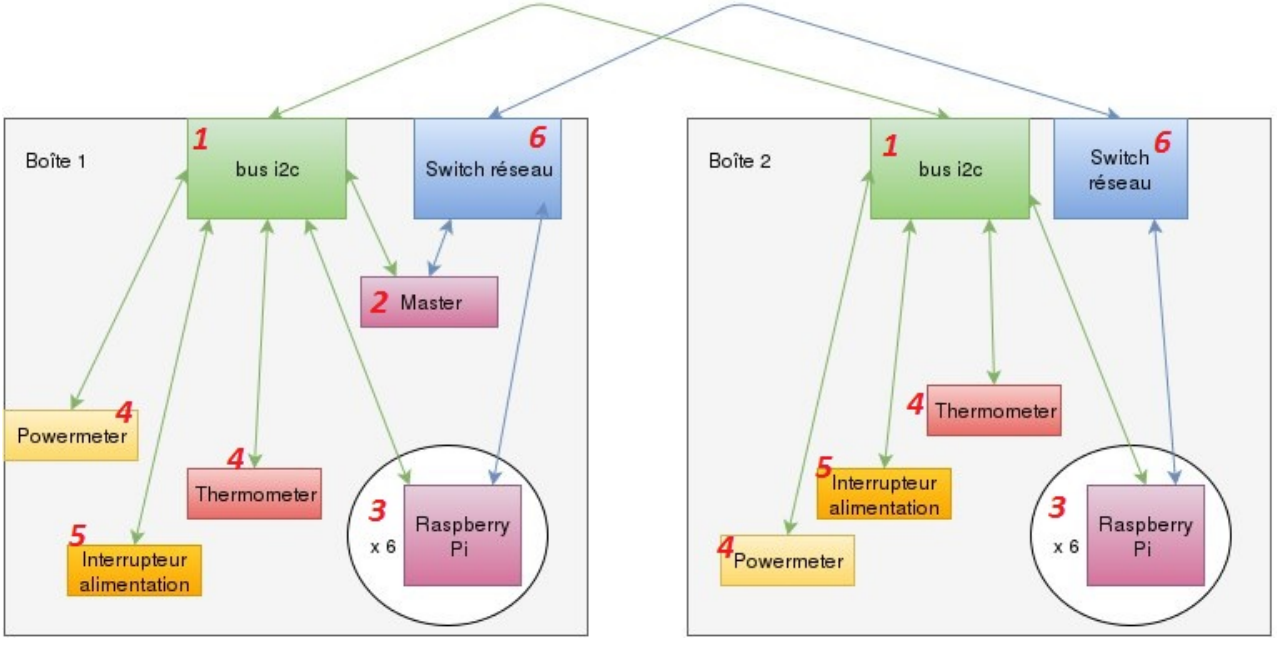

FIGURE 1 – Architecture matérielle

Ainsi, le but était de réaliser une interface permettant d'interagir avec tous ces éléments. Nous présenterons dans ce rapport tout d'abord le contexte et le produit que nous devions réaliser, en définissant les différents termes propres au projet et en détaillant ce dernier dans son ensemble. Nous nous attarderons ensuite sur toute la gestion du projet que nous avons mis en place, en évoquant également quelques problèmes que nous avons pu rencontrer. Enfin, nous finirons par détailler le travail effectué selon les différentes parties du projet, et le produit final obtenu.

1 Présentation du projet

Dans ce contexte particulier, notre but était ainsi de réaliser un cluster de raspberries. Pour présenter convenablement le travail effectué, il convient ainsi tout d'abord de bien présenter le projet et ses enjeux.

1.1 Besoins et but

L'intégralité du projet s'articule autour du grape cluster, désignant des rapsberries interconnectés représentant un réseau, distinguant un maître et ses esclaves. Ce dernier est principalement destiné (dans un premier temps) aux étudiants, pour pouvoir réaliser des Travaux Pratiques en réseau, cloud ou big-data, et aux chercheurs dans l'optique de réaliser des expérimentations. Il serait de ce fait opportun d'offrir à ce système un moyen d'interaction et de contrôle. C'est de ce constat que découle directement le besoin principal du projet, qui sest donc la nécessité de réaliser une interface pour contrôler ce "Cluster". Celle-ci aura pour but d'offrir plusieurs fonctionnalités à l'utilisateur, telles que lancer des tâches sur les esclaves, contrôler le réseau, ou obtenir de informations sur la carte telle que la température ou la consommation électrique.

Ce besoin central va donc induire les objectifs de ce projet. Le but sera de réaliser une interface graphique que l'utilisateur pourra lancer simplement. Depuis cette interface, il doit pouvoir envoyer des ordres ou des requêtes au maître. Il faut ainsi pouvoir mettre en place un module du maître qui communiquera avec l'interface. Enfin, il faut que ce dernier module puisse également communiquer avec les esclaves, pour demander des informations ou donner des ordres, et récupérer la réponse, qu'il pourra alors retransmettre à l'interface. L'utilisateur pourra ainsi obtenir ce qu'il souhaite directement via l'interface, et procéder aux expériences réseaux comme il le souhaite.

Ainsi, le projet va nécessiter le développement de plusieurs modules qui communiqueront entre eux. L'architecture globale attendue peut alors être représentée comme le montre le schéma 2. Ces différentes parties vont constituer le coeur du projet, et leur disparité constitue également la spécificité même du projet.

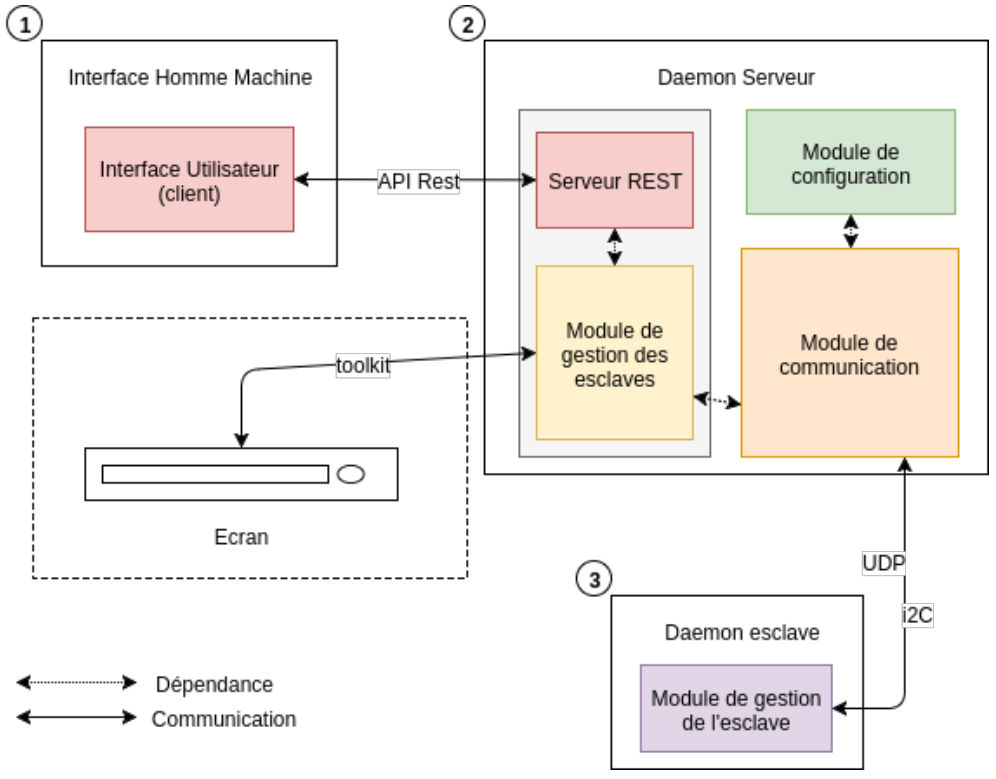

FIGURE 2 – Modélisation du fonctionnement du projet

1.2 Un projet modulaire

Ce projet, du fait de ses besoins, va en effet se séparer naturellement en plusieurs parties distinctes qui s'inter-connecteront. Il convient alors de bien définir chacune d'entre elles, en précisant leurs fonctions, leurs buts, les technologies nécessaires et leur limites.

Initialement, il était question de trois parties principales, concernant le daemon (processus d'exécutant en arrière plan) du maître, l'interface graphique et le daemon de l'esclave. Nous avons donc ces trois modules différents :

— **Module du maître :** L'enjeu est d'implémenter un module permettant de contrôler le maître. Ce code, qui sera en python, doit permettre d'envoyer et recevoir des requêtes pour le client, et de communiquer avec les esclaves pour collecter des informations.

— **Module de l'esclave :** Dans cette partie, on a besoin d'une part d'un daemon permettant d'interagir avec le cluster en récupérant des informations, et d'autre part de réaliser la communication en I2C, particularité essentielle du projet.

— **Module de l'interface graphique :** Ce dernier module devra donc fournir une interface pour que l'utilisateur puisse interagir avec les raspberries à sa guise. Développé en Javascript, son rôle est donc d'interroger le daemon du maître et d'afficher les informations.

A ces trois modules principaux se sont rajoutées deux autres parties non centrales : un script construisant une image pour nos esclaves, et un serveur TFTP. Ces deux points sont assez extérieurs au coeur du projet, car ce sont des développements dont l'impact est non négligeable, mais non classable au même titre que celui du développement du Daemon maître par exemple, puisque annexe par rapport à la ligne principale.

— **Script de "génération" d'image :** Le but est ici d'enrichir l'image d'un système comme Raspbian, afin de faciliter par la suite les actions répétitives ou fastidieuses. Ce script sera un script shell qui permettra une automatisation et une certaine modularité dans l'utilisation du grape cluster. Ainsi, le script pourra par exemple installer des paquets, et pré-configurer le raspberry.

— **Serveur TFTP :** Au lancement d'un raspberry, nous souhaitons vérifier si ce dernier possède bien la dernière image de son système d'exploitation. Via le serveur TFTP du maître ainsi qu'une image minimale, ce dernier vérifie et remplace donc son image si besoin est.

Ces différents modules vont complètement définir la manière d'aborder le projet, que ce soit en terme technique avec les technologies utilisées, ou en terme pratique en ce qui concerne la gestion de projet et la répartition des tâches.

1.3 Technologies utilisées

Nous avons utilisé afin de mener à bien ce projet des outils et des technologies variées. Tout d'abord concernant le projet en lui-même, nous avons programmé avec trois technologies, langages différents. Toute la partie correspondant au Daemon maître a été réalisée avec le langage *Python*. Son côté interface Web notamment (incluant l'API Rest) se repose sur le framework *Flask* et le *templating engine* nommé *Jinja2*. Ceci dit pour réaliser cette dernière, il nous a fallu utiliser les langages basiques du Web comme le *CSS*, *HTML* et *Javascript*. Afin de faciliter le développement nous avons de plus utilisé le framework Javascript *JQuery*, notamment pour ses appels *Ajax* plus accessibles. Côté rendu Web, nous avons utilisé le style standard proposé par le framework CSS *BootStrap*. Finalement toute la partie Daemon esclave a elle été intégralement faite avec le langage *C*, et quelques fonctions de la bibliothèque *GNU*. À part, le script de génération d'images est lui un script shell réalisé en *Bash*.

Pour la programmation en elle-même, nous nous sommes servis de plusieurs outils pour écrire le code : l'IDE [1] édité par *JetBrains*, *PyCharm* qui nous a particulièrement aidé pour programmer et débugger Python, ainsi que Flask. Pour la programmation en Shell, en C et Web nous avons utilisé selon les préférences personnelles de chacun *Visual Studio Code* et *Sublime Text* dont les systèmes d'extensions permettent une modularité non négligeable.

Concernant nos outils d'équipe, nous avons utilisé *GitHub* pour nous permettre de mettre en commun le code produit et plus particulièrement dans l'objectif de le versionner. L'outil *Trello* dont nous détaillerons l'utilité plus bas nous a permis une certaine organisation en "tâches" pour la programmation, tandis que *Telegram* nous a fourni un moyen rapide et facile de communiquer de manière "instantannée". Pour terminer, nous avons rédigé ce rapport en LaTeX sur la plateforme collaborative *ShareLatex*. La figure ci-dessous représente succintement à partir de leurs logos respectifs toutes les technologies que nous venons de nommer ici.

FIGURE 3 – Logos des différentes technologies utilisées

1. Integrated Development Environment : Environnement de Développement Intégré

2 Gestion de projet

Dans cette première partie nous allons nous attacher à présenter nos différents modes de gestion de projet ainsi que les outils que nous avons pu utiliser afin de nous y aider. En effet nous avons l'habitude de travailler par équipe de 3 ou 4, mais pour ce projet il a fallu travailler avec 7 développeurs et 1 superviseur ce qui s'est avéré être bien différent et plus complexe que ce que nous pensions. Nous avions donc décidé de mettre en place plusieurs méthodes.

2.1 Une séparation des tâches et le travail d'équipe

Tout d'abord, il convenait, du fait de la disparité des modules à développer, de réaliser une séparation des tâches pertinente. Comme nous avons pu le présenter précédemment, ce projet se sépare naturellement en trois parties principales, nécessitant chacune plusieurs tâches afin d'être menées à bien. De plus, d'autres plus petites parties s'ajoutent à cela. Ainsi, nous avons dès le début du projet réparti ces différentes parties entre nous, afin qu'au moins deux personnes travaillent sur chacune de ces dernières, et puissent ensuite se séparer les tâches relatives à cette partie pour travailler efficacement.

Cette efficacité devait aussi passer par une gestion du code propre et facilitée. Ainsi, un outil de versionning tel que Git s'est révélé plus que nécessaire. Grâce à cet outil, nous avons pu versionner le code et travailler efficacement en parallèle, en facilitant la fusion de deux travaux. De plus, nous avons utilisé le concept de branches pour pouvoir séparer les différentes parties du projet et rendre cohérent le dépôt. Chaque partie a donc été développée indépendemment. Néanmoins une étape de fusion a été nécessaire, et cela s'est révélé plus compliqué que prévu. Une première partie a été de lier partie Maître et Web en établissant la communication entre les deux, et permettant à l'interface Web d'envoyer des demandes au Maître qui pouvait alors répondre. Cette étape a été plutôt naturelle de part l'architecture REST, ainsi que l'utilisation de Flask. Une deuxième étape de cette fusion a été de lier l'i2c et l'UDP afin que l'esclave ait un seul deamon en action écoutant d'un côté l'UDP et de l'autre côté l'i2c. Cette fusion a donc été un peu plus complexe que la première car nous avons du gérer les fichiers communs.

Avec du recul nous nous sommes rendu compte que découper les tâches peut se révéler être une action nécessaire pour certaines parties bien distinctes mais cela peut être aussi motifs de problème lors de l'étape de fusion. Par exemple pour l'UDP et l'i2c une méthode plus agile aurait été nécessaire avec dès le départ un fichier permettant de passer de l'UDP à l'i2c facilement.

Enfin, un autre aspect de ce projet qui s'est révélé primordial a été la communication. En effet, bien que les parties soient clairement séparées, le fait de communiquer entre les différentes parties pour s'organiser et assurer une connexion plus facile ensuite a été important. C'est un aspect que nous avons un peu négligé au début et qui nous est ainsi apparu de plus en plus nécessaire au cours du projet. Ce besoin de communication efficace nous a donc poussé à établir plusieurs solutions pour s'assurer de ne plus reproduire l'erreur, en mettant en place des points réguliers pour faire part de l'avancement de chacun, mais aussi en utilisant des outils comme Trello, qui a eu une double fonction, la communication, et la gestion de l'agilité.

2.2 Un développement semi-agile

Cette caractéristique agile a en effet été un aspect très important du projet lors du développement. Après avoir fait une première phase plutôt classique de conception de projet, nous avons lors du développement essayé de basculer vers un développement plutôt agile.

2.2.1 Une totale agilité compromise

Dans un premier temps, nous avions donc pour but d'utiliser les méthodes agiles telles que l'on a pu les étudier. L'idéal était donc de parvenir à un flux de travail (workflow) se séparant en plusieurs

sprint durant lesquels plusieurs tâches seraient à faire, avec un scrum-master, dont le rôle serait de superviser et gérer le déroulement du développement.

Cependant, du fait des conditions et du contexte qu'est le projet de PFA, il n'était pas si simple de mettre en place une réelle méthode agile telle qu'elle est définie formellement. Nous avons ainsi établie des méthodes de développement en s'inspirant au mieux des méthodes agiles tout en nous adaptant aux contraintes de la situation. Ainsi, plus de scrum-master et pas de réels sprint, mais une équipe qui s'auto-gère et distribue les tâches, gère l'avancement et fait le bilan en se réunissant à des périodes régulières données. Notre superviseur n'a pas pu non plus endossé le rôle de scrum-master du fait de son rôle de correcteur, mais a apporté une aide précieuse pour cette gestion pseudo-agile en nous faisant prendre conscience de la difficulté de mettre en place une méthode, et du besoin de rigueur que cela demande. Nous avons pu au cours du projet étayer la méthode et l'améliorer, en accordant un temps non négligeable à l'établissement d'une certaine rigueur dans la méthode.

2.2.2 Un outil en particulier

Nous avons malgré tout utilisé des outils caractéristiques au développement agile, et particulièrement Trello, un outil de gestion en ligne, qui permet à l'image des projets en agile, d'avoir un mur virtuel partagé entre tous les membres de l'équipe sur lequel on peut ajouter les différentes tâches à effectuer. On peut pour chacune d'entre elles spécifier les membres, la difficulté, des commentaires ou d'autres informations utiles. La gestion "agile", la répartition des tâches et la visualisation de l'avancement a ainsi pu se faire beaucoup plus aisément, d'autant plus dans notre cas où il n'y avait pas de scrum-master et donc où tout le groupe devait suivre l'avancement continuellement. Les figures ci-dessous sont des captures d'écran de notre "mur" Trello à un moment donné du projet.

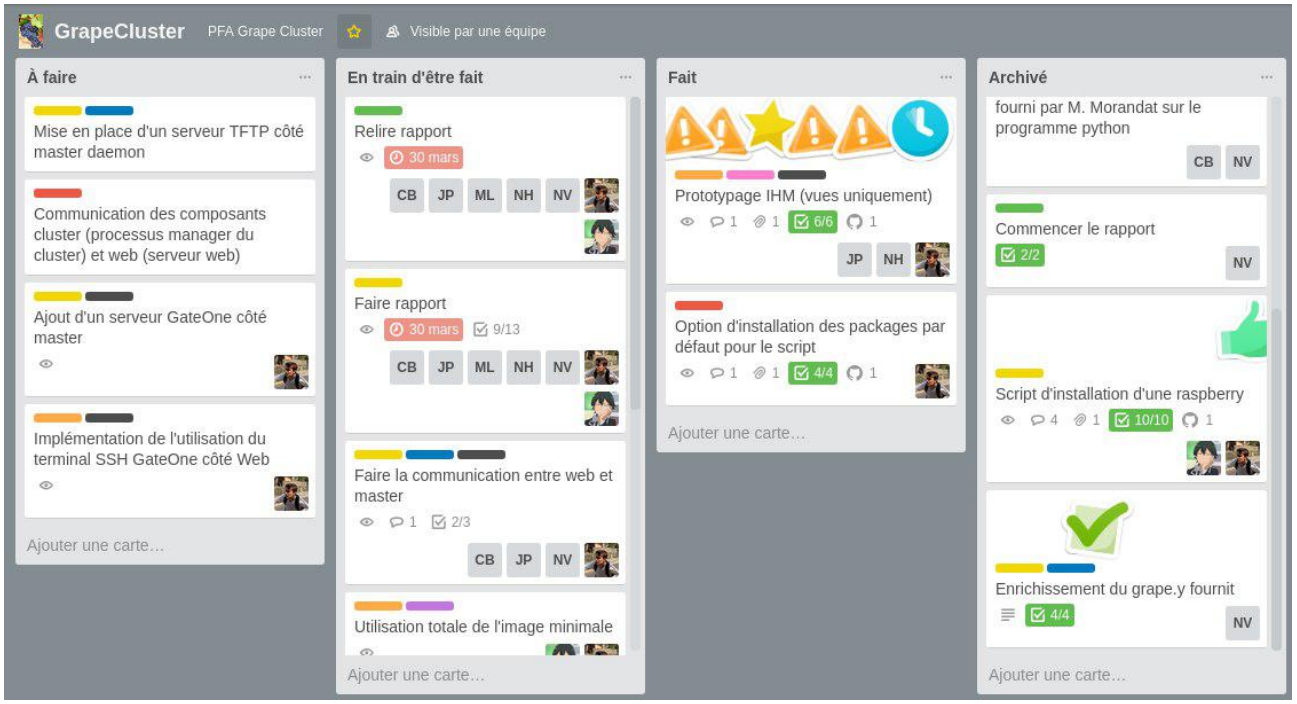

FIGURE 4 – Prise d'un instantanné de notre tableau Trello

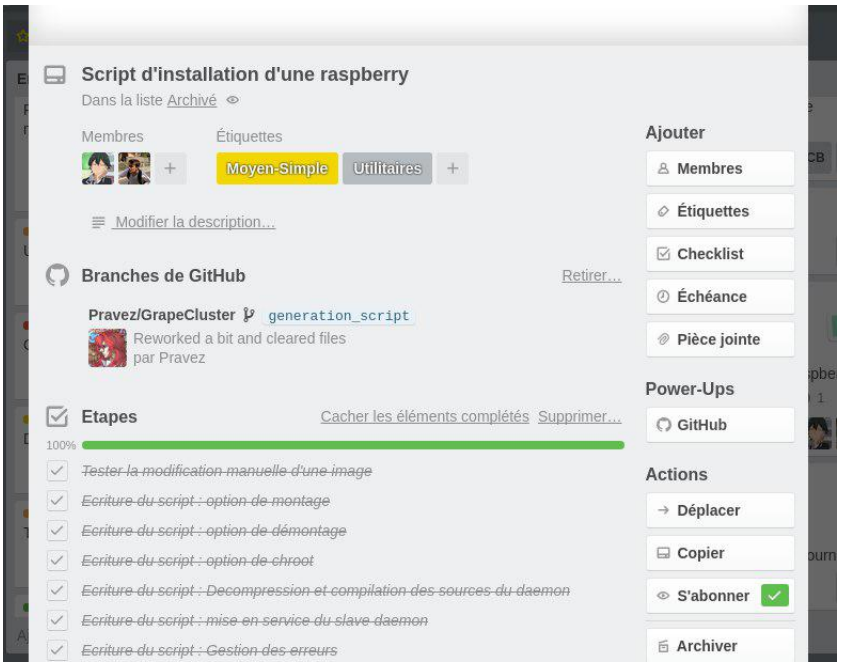

FIGURE 5 – Description d'une tâche Trello

Trello n'est pas le seul sur le marché de ce genre d'outils. Par exemple, Wrike est un autre outil de développement Agile ayant les mêmes idées que Trello, mais s'axant davantage sur la communication interne à l'outil avec des salons de discussions et autres. Le fait est que nous ne cherchions qu'un simple outil nous permettant de partager des tâches sur un tableau basique, sans chercher de réelles fonctionnalités développées. Nous avons davantage souhaité essayer le développement de ce type en tant que tel, que chercher un outil optimal, et Trello nous a donc parfaitement convenu.

Nous avons ainsi pu parvenir à une gestion pseudo-agile, qui aurait pu être peut-être plus rigoureuse par certains aspects en fixant des durées fixes de sprint, mais qui nous a parfaitement convenu pendant le développement et nous a permis d'affiner un peu notre expérience en méthodologie, et de mener à bien plus efficacement le projet. De plus l'organisation de Trello en "Power-Ups" nous a permis de lier notre mur au dépôt GitHub, de telle sorte que nous avons pu rattacher des *commits* ou des *branches* à des tâches particulières du tableau.

2.3 Problèmes rencontrés

Ainsi, comme nous avons pu succinctement l'évoquer précédemment, nous avons parfois rencontré quelques difficultés au cours du projet. D'une part, de par les contraintes même entraînées par le PFA, une part non négligeable du travail doit être réalisée en dehors des horaires fixées initialement, ce qui a pu nous poser des problèmes d'ordre organisationnel. En effet, il n'a pas toujours été facile de se coordonner pour travailler parallèlement sans pour autant pouvoir travailler au même moment ou sur les mêmes parties. Nous avons du apprendre au fur et à mesure à développer le travail d'équipe au delà-même du cadre des séances scolaires pour établir une organisation plus saine et mener à bien notre objectif.

Ensuite, nous avons également pu rencontrer des difficultés inhérentes au projet en lui même, et notamment au niveau de la partie matérielle du projet. En effet, la partie I2C, beaucoup plus technique que les autres car nécessitant des branchements précis pour marcher et être utilisée, et la partie sur laquelle la durée de temps passée à résoudre des problèmes a été la plus importante. Beaucoup plus éloignés de ce qu'on a l'habitude de faire et relativement compliqué à appréhender aux premiers abords, les quelques nécessités électroniques et techniques nous ont donc fait souvent obstacle, et la résolution de ces problèmes nous a réellement permis d'avancer plus sereinement par la suite. Cet aspect, bien

qu'étant donc un des plus difficile du projet car il entraînait des contraintes liées au matériel, ce qui n'est pas forcément usuel dans notre formation, a aussi été l'un les plus intéressants et formateur.

Ces différents obstacles, plus ou moins important et compliqué à résoudre, ont tous eu certaines conséquence sur le projet en lui-même. Au delà même de la gestion et des prévisions effectuées, il est intéressant de regarder la réalité du travail produit, en particulier au niveau de sa cadence et de la temporalité du projet. En effet, malgré toutes les anticipations et prévisions sur le planning que l'on peut faire en début de projet, il est certain que des problèmes inattendus risquent de surgir et que cela impacte notre rythme global. Cet aspect particulier d'un projet a clairement pu être expérimenté dans le cadre de notre PFA, et la cadence de travail n'a ainsi pas toujours été homogène et régulière, principalement à cause d'obstacles que l'on a pu rencontrer. Nous avons ainsi plusieurs fois oscillé entre période de résolution de problème ou de contournement de risque et période de développement et d'ajout de fonctionnalités essentielles au produit final. Notre flux de travail s'est ainsi révélé relativement différent de celui que nous avions prévu lors de la phase de conception, comme on peut le voir sur le diagramme de Gantt 7 représentant l'avancement réel qu'a connu le projet au cours de son développement pour chaque tâche 6. Nous sommes malgré tout parvenu à limiter les risques pour assurer le rendu final.

Note : Le diagramme de Gantt a été découpé en deux parties pour plus de lisibilité.

Installation d'un système sur un Raspberry	23/01/17	
Communication I2C	13/03/17	05/04/17
Prototypage du protocole de communication	27/01/17	
implémentati...	10/02/17	16/03/17
script d'installation	23/01/17	06/04/17
IHM	08/02/17	06/04/17
installation d'une image sur un raspberry	08/02/17	05/04/17
Script d'installation d'une image	08/02/17	24/02/17
Daemon master	09/02/17	06/04/17
Serveur REST	16/02/17	05/04/17
Création des routes POST	16/02/17	05/04/17
Création des routes GET	16/02/17	24/03/17
Implémentation de la récupération des données du Cluster	09/02/17	03/04/17
mise en place d'un serveur TFTP	31/03/17	06/04/17
Implémentation de la communication avec un slave	10/02/17	05/04/17
abstraction communication I2C/UDP	10/02/17	02/03/17
implémentation de la communication UDP	10/02/17	03/03/17
implémentation de l'I2C	13/03/17	05/04/17
Intéraction avec les slaves	10/02/17	16/03/17
Intéraction UDP	10/02/17	16/03/17
Intéraction I2C	27/03/17	06/04/17
IHM	08/02/17	06/04/17
Prototypage IHM	08/02/17	05/04/17
squelette de l'IHM	08/02/17	24/02/17
prototypage des différentes vues	15/02/17	05/04/17
Routes de l'API REST	20/03/17	06/04/17
Récupération des données du master (GET)	20/03/17	24/03/17
Demandes de commandes (POST)	24/03/17	06/04/17

FIGURE 6 – Liste des tâches représentées dans le Gantt

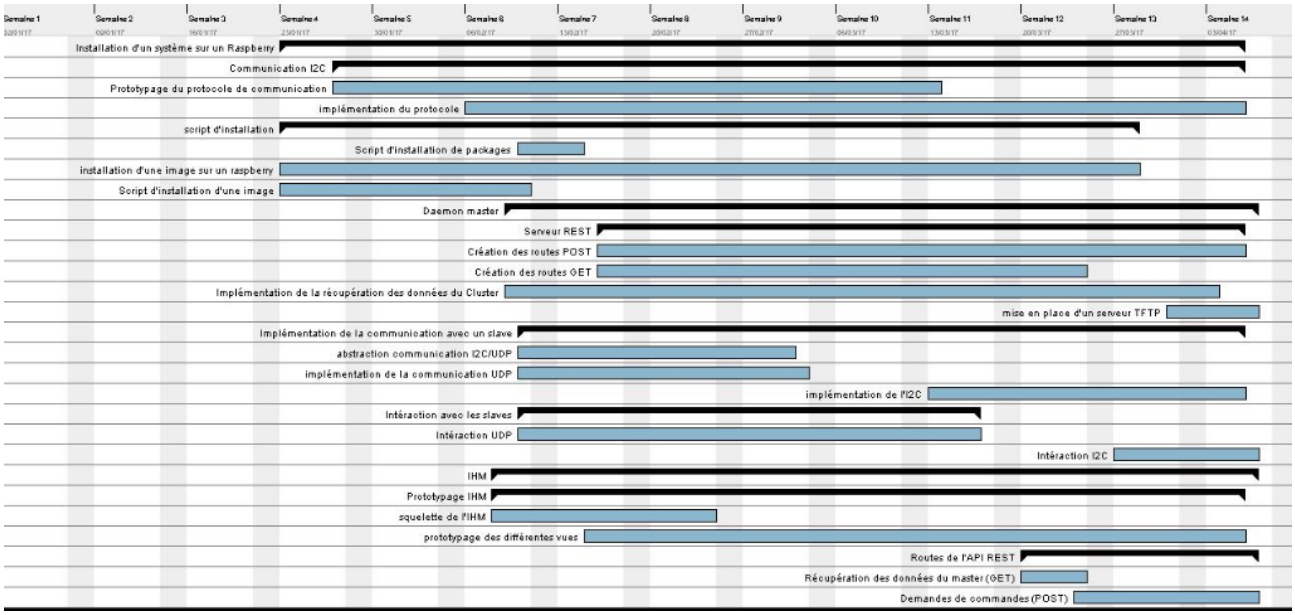

FIGURE 7 – Gantt représentant le flux de travail réellement réalisé

On peut constater que certaines parties sont très étendues dans le temps. Cela s'explique de deux manières : par des ajouts ou des corrections au fur et à mesure que le projet évoluait et par des difficultés rencontrées évoquées plus tôt. Si on prend l'exemple du script d'installation, un script viable pour une image précise a été assez rapidement implémenté mais des ajouts de différentes options font que la répartition se retrouve être plus étendue.

Malgré les différentes difficultés que nous avons pu rencontrer, nous sommes finalement parvenus à un résultat satisfaisant. Cette gestion des problèmes et des corrections a principalement été possible grâce aux réunions régulières qui ont permis de nous tenir mutuellement au courant des problèmes de chacun pour discuter des solutions à mettre en place, et continuer à avancer en apprenant des erreurs et des complications croisées.

3 Résultats finaux

Après avoir passé la phase de conception et à travers l'usage de toutes ces méthodes de gestion, nous avons donc finalement développé chaque module afin de répondre aux différents besoins. Chaque partie a ainsi impliqué quelques éléments intéressants à présenter, mais aussi parfois des problèmes. Leur réalisation s'est ensuite terminée par la connexion entre chaque élément pour former un tout cohérent qu'est le produit final.

3.1 Le Daemon du maître

Ce module consacré au développement d'un Daemon pour le maître a été un des plus important. En effet, c'est lui qui fait la liaison entre l'interface graphique et les esclaves. S'il est possible d'obtenir quelque chose d'exploitable en enlevant l'interface (où on aurait alors tout de même un maître qui peut interagir avec les esclaves et leur envoyer des ordres) ou en enlevant les esclaves (où on aurait une interface qui peut tout de même afficher les données du maître), le retrait de ce module de maître rend les autres modules inutilisables sans rajouter autre chose. Ainsi, il a fallu apporter une attention particulière au code pour qu'il soit à la fois valide, efficace et réutilisable. Les enjeux de ce Daemon étaient ainsi doubles. D'un coté, nous devions établir la liaison avec l'interface graphique à l'aide de routes REST et du framework Flask, et de l'autre, établir la liaison avec les esclaves et communiquer avec eux à travers de sockets UDP.

3.1.1 Liaison avec interface graphique

Le premier travail a ainsi été de faire en sorte que notre module puisse communiquer avec l'interface. Selon les directives du client, les différentes informations et les ordres sont supposés être émis à l'aide de requêtes HTTP via une architecture REST dont les routes permettent l'utilisation du cluster. Pour cela, nous avons choisi d'utiliser le framework Flask, offrant en Python tous les outils pour définir et utiliser des routes REST. Cette architecture va permettre de définir différentes routes, qui correspondront chacune à un traitement que l'on souhaite effectuer. Pour schématiser et mieux comprendre le concept de l'architecture REST, on peut observer l'exemple présentant le principe de nos routes sur ce projet sur la figure 8.

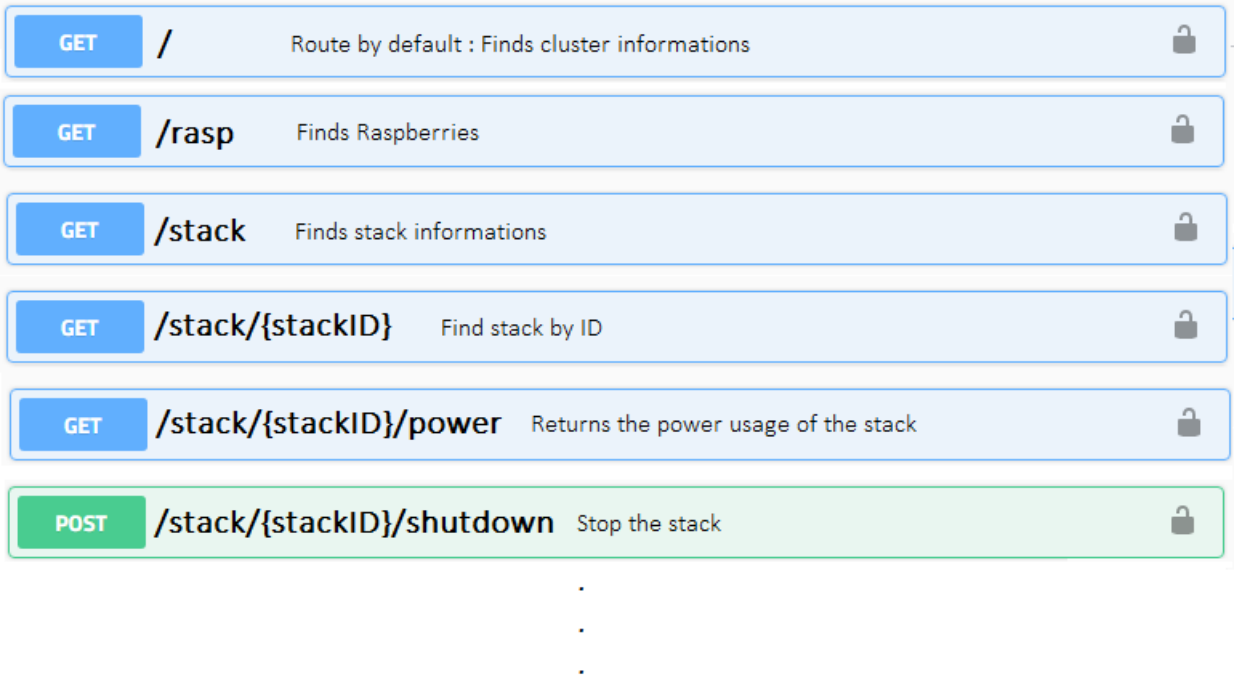

FIGURE 8 – Représentation partielle de l'architecture REST

On voit que l'architecture part d'une racine "/", une route par défaut, qui correspondra, sur l'interface, à la page d'accueil. Arrive ensuite les ramifications qui vont permettre de spécifier ce dont on veut parler. Ici, les Stacks, désignant une borne de cluster dans le code, ou les Raspberries. Cette route permettra d'obtenir des éléments constituant le premier type de ressource que l'on peut vouloir récupérer : des collections. Ici donc, la collection des bornes ou des Raspberries d'une borne. Ensuite, d'autres ramifications deviennent plus précises et demande des informations plus spécifiques, en précisant un id. On accède ainsi à des ressources du deuxième type : des instances>. On veut alors n'afficher les informations que d'une borne précise. Enfin, une nouvelle ramification permet de rentrer encore plus dans le détail en demandant des informations précises, ou désigne même des ordres. Par exemple, la route "/Stack/<id>/power" qui va permettre de demander la consommation d'énergie d'une borne, qui correspondra à une méthode GET dans l'interface. Au contraire, la route "/Stack/<id>/shutdown" correspondra à une méthode POST et enverra l'ordre à une borne de s'éteindre.

En python, cette architecture se traduit très facilement avec la bibliothèque qu'est Flask, et on peut ainsi aisément définir les différents traitements associés à chacune des routes. Prenons pour exemple la route "/Stack/<id>/shutdown" permettant d'éteindre un Raspberry donné. On peut voir comment va se traduire cette route dans l'implémentation de l'architecture REST en python sur la figure 9. On définit d'abord la route en elle-même, en précisant au passage ici qu'il s'agit d'une méthode POST. On définit ensuite une fonction associée à cette route REST qui sera appelée lors de l'accès correspondant. Elle prendra automatiquement en paramètre l'entier présent dans la route définie une ligne plus haut. On peut ainsi ensuite décrire le comportement à adopter, puis faire appel à notre application Json pour la réponse.

```python
@app.route('/stack/<int:id>/shutdown', methods=['POST'])
def routeShutdown(id):
    stack = daemon.get_master().get_stack(id)

    if stack is None:
        response = 0
    else:
        response = 1

    return app.response_class(
        response=json.dumps({'response': response}),
        status=200,
        mimetype='application/json')
```

FIGURE 9 – Exemple de code implémentant une route REST

C'est ainsi que chacun des traitements vont pouvoir être reliés à l'interface, correspondant à un noeud de cet arbre. Il est intéressant de voir que chacun de ces traitements peut être traduit par une phrase en partant de la racine, et rendant ainsi assez intuitive l'utilisation de cette architecture. Par exemple, la route "/Stack/<id>/power" correspondrait à la phrase : "Dans mon architecture actuelle, je veux afficher à propos de la borne (/Stack) d'identifiant b (<id>), la consommation d'énergie (/power)". Chacune de ces routes est ainsi associée à une méthode de notre Daemon, qui est alors appelée à chaque fois que l'interface active la route correspondante.

C'est ainsi que le Daemon du maître va pouvoir communiquer facilement avec l'interface et recevoir facilement les requêtes de celui-ci.

3.1.2 Liaison avec le Daemon des esclaves

D'autre part, l'autre enjeu de cette partie est de réaliser la liaison avec les esclaves, pour pouvoir leur envoyer des requêtes ou des ordres. Le problème était ici de pouvoir communiquer grâce à des sockets, mais sans protocole fixe. En effet, il était important de permettre la communication à la fois en I2C, mais aussi en UDP pour les messages beaucoup plus gros ou des cas particuliers de communication. Nous avons répondu à cette contrainte matérielle par une solution logicielle à travers l'architecture que nous avons construit.

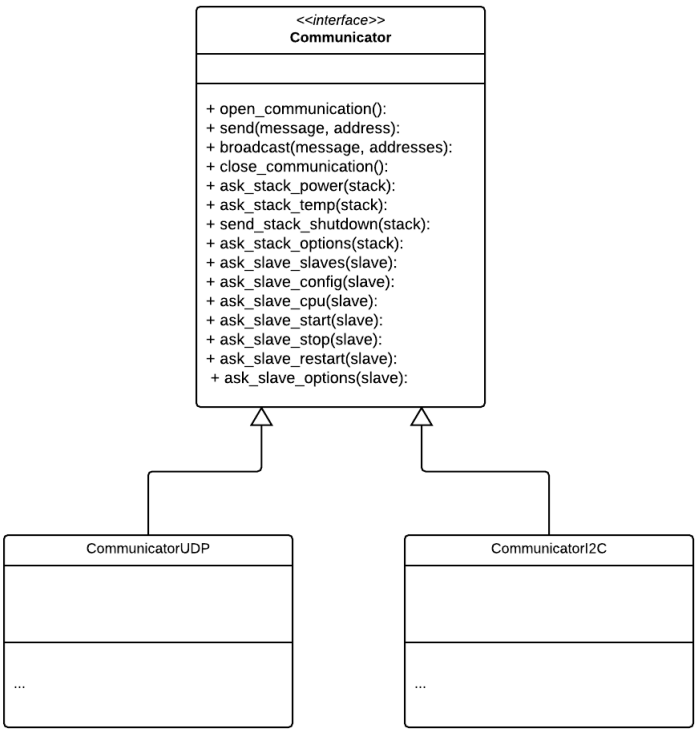

FIGURE 10 – UML du module Communicator ajouté

En effet, nous avons créé une interface Communicator comme présentée sur la figure 10, permettant de définir les différents traitements que l'on doit pouvoir effectuer, tout en faisant abstraction du protocole utilisé. Il suffit ainsi ensuite de définir des classes implémentant cette interface, tel que CommunicatorUDP, qui va permettre d'associer à chacun des traitements définis une solution avec le protocole UDP. En plus de rendre le code plus clair et modulable, cela le rend également plus évolutif, puisqu'il est de cette façon très facile d'ajouter une nouvelle communication via un autre protocole en créant une nouvelle classe. Ainsi nous avons également une classe CommunicatorI2C qui implémente les méthodes de l'interface Communicator et permet une parfaite transparence entre UDP et I2C.

En UDP, le maître reçoit des informations supplémentaires, impossible à recevoir en I2C avec lequel la quantité de données reçue a été fixée à 4 octets de par les contraintes du protocole. Ceci conforte l'idée de généraliser les deux communications avec une interface tout en permettant de gérer les détails d'implémentation spécifiques à chacune dans les classes les réalisant. Ainsi en UDP, les ordres envoyés correspondent aux mêmes qu'en I2C, tout deux chargés à l'aide du fichier orders.txt à la racine du dépôt et parsé ensuite par les deux Daemon. L'envoi des messages UDP suit donc un codage strict, précisant lors du retour du slave le type de requête auquel il répond. Les 4 octets de l'I2C ne permettent pas ce niveau de détail et n'est pas nécessaire, là ou en UDP le caractère asynchrone de la communication fait qu'en acccumulant les requêtes sans réponses il faut utiliser un système permettant de repérer les réponses et de gérer les requêtes en cours.

```
12  class Daemon(Thread):
13      def __init__(self, ip_address):
14          super(Daemon, self).__init__()
15
16          self.__master = Master(0, "00:00:00:00:00:00", ip_address, "0", 0)
17
18          self.__master.add_stack(Stack(0,0))
19
20          self.__udp_comm = CommunicatorUDP(42666, self.__master.get_ip_address())
21
22          self.__udp_comm.open_communication()
23          print("Master listening on {}:{}".format(ip_address, 42666))
24
25      def run(self):
26          while True:
27              print("Broadcasting cpu request")
28              self.__udp_comm.broadcast("1;", self.__master.get_cluster_ip_addresses())
29              data, addr = self.__udp_comm.receive(1024)
30              print("received message: {} from {}".format(data, addr))
31              if data == b"configure":
32                  stack = self.__master.get_stack(0)
33                  new_slave = Slave(0, "AA:AA:AA:AA:AA:AA", addr[0], "0", len(stack.get_pi_devices()))
34                  self.__master.get_stack(0).add_pi_device(new_slave)
35                  self.__udp_comm.send("0;" + addr[0] + ";", new_slave.get_ip_address())
36                  print("Configured new slave of ip_addr {}".format(addr[0]))
37
38              elif data[:4] == b"cpu:":
39                  self.__master.get_slave_by_ip(addr[0]).cpu_usage = data[4:]
40                  print("Received cpu_usage from slave {}, updating value".format(addr[0]))
41
```

FIGURE 11 – Exemple d'utilisation d'un Communicator dans un code

Ainsi, nous pouvons voir sur la figure 11 un exemple d'utilisation de ce Communicator dans le cadre de notre Daemon. On peut voir que la communication est grandement simplifiée. Par exemple, à la ligne 22, on peut ouvrir facilement la communication, sans se soucier du protocole utilisé. De même, à la ligne 34, l'envoi de message nécessite simplement le passage du message à envoyer en paramètre, et le polymorphisme permet d'abstraire le comportement spécifique à chaque protocole pour chaque classe.

De plus, on peut également voir grâce à cet exemple le principe général pour lancer une communication du côté du maître. Une initialisation est d'abord nécessaire, permettant à la fois de définir les principaux attributs du maître tels que ses différentes adresses (ip, mac et I2C), d'ajouter ce maître à la borne représentant le cluster sur lequel on travaille, et d'initialiser la connexion. Ensuite, on peut démarrer la communication en tant que telle. Cet exemple présente en l'occurrence un envoi en broadcast, et donc un envoi à tous les esclaves qui se trouvent sur la borne actuelle. Ici, on récupère toutes les adresses des éléments formant le cluster, avant d'envoyer le message, et de se mettre en attente de la réponse. Une communication avec les esclaves qui est donc simplifiée au maximum, pour rendre le code facilement compréhensible et réutilisable.

Le Daemon du maître parvient ainsi à répondre à tous les besoins qu'il devait combler, en créant une communication avec les esclaves grâce à une implémentation de la classe Communicator, lui permettant d'envoyer et de recevoir des messages simplement, et en recevant les requêtes de l'interface grâce aux routes REST, qui lui permettent de savoir quelle action est demandée et quel message il doit envoyé au Daemon esclave.

3.1.3 Les différences avec le cahier des charges

Le modèle des données du master représente le cluster, principalement grâce à la classe Master et la classe Slave qui héritent toutes les deux de PiDevice. Ainsi on stocke les diverses informations propres à chaque Raspberry (adresse IP, adresse I2C, ...) mais aussi les dernières données reçues comme le taux d'usage CPU. Ceci permet ensuite au serveur Web d'utiliser les getters associés pour mettre à jour la vue, ou à l'API Rest de desservir les dernières valeurs rafraîchies.

Pour permettre une intégration au serveur, le Daemon Master implémente désormais l'interface Thread et ses méthodes start() et run(). Ainsi, la méthode run() contient la logique applicative du master. Le serveur instancie ce thread et appelle les fonctions du master pour récupérer les informations

du cluster (get_Stacks() notamment, puis get_pi_devices() et enfin get_ip_address() et autres). Le Daemon est quand même capable de tourner sans le serveur Web, pour cela un main très simpliste est fourni.

La solution finalement apportée n'est pas exactement celle que nous avions planifiée dans le cahier des charges lors de la phase de conception. En effet, nous avions défini une architecture logicielle globale permettant de compléter le code déjà mis en place et de répondre au besoin. Cependant, c'est durant le développement en lui-même que nous avons pu effectivement mettre en application cette démarche, et que nous avons pensé à améliorer la solution prévue en y ajoutant l'abstraction de la communication avec l'interface Communicator. Tout cet aspect de notre code s'est donc rajouté durant le développement, et fait que l'architecture logicielle finale diffère de celle présentée dans le cahier des charges.

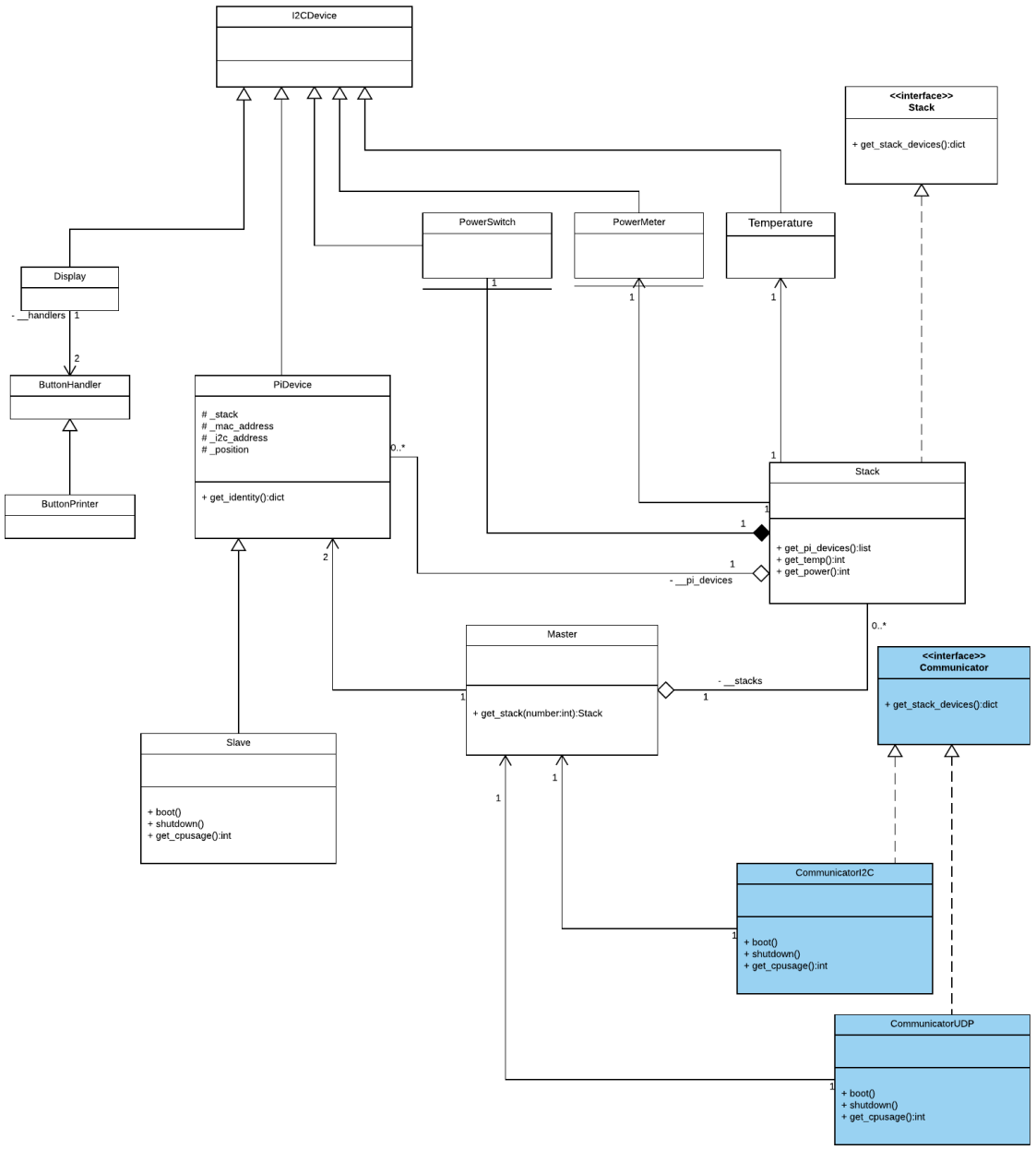

FIGURE 12 – UML représentant l'architecture logicielle, avec en bleu les différences avec les prévisions

Comme on peut le voir sur l'UML final 12, plusieurs classes (repérées en bleues) se sont ajoutées à

la structure logicielle finale. En revanche, certaines ont disparu, à l'image des trois interfaces que nous avions pensé mettre en place pour abstraire la définition des *PowerMeter*, *TemperatureSensor* et le *PowerSwitch*. En effet, nous avions dans un premier temps pensé qu'il serait utile d'abstraire la définition de ces classes à l'aide d'interfaces pour soustraire les contraintes matérielles des modèles de capteurs utilisés. Ainsi, nous voulions définir des classes abstraites dont hériteraient des classes spécifiant l'utilisation des modèles utilisés dans le Grape Cluster actuel, tout en offrant ainsi la possibilité de créer plus tard d'autres classes définissant d'autres modèles plus facilement. Malheureusement, nous avons du renoncer à cet ajout, car cela changeait la structure même du logiciel dans des proportions bien plus grandes que ce que nous pensions. En effet, ces différents capteurs héritant de la classe *I2CDevice*, il fallait que les interfaces traduisent ce lien dans leurs définitions également. Or, ce lien devenait impossible à faire entre les interfaces souhaitées et l'*I2CDevice* qui n'est pas une interface. Nous avons ainsi du une nouvelle fois nous adapter, et choisir la solution la plus adéquate pour répondre au besoin le mieux possible.

De plus, nous avons également changé les routes REST que nous avions déjà définies dans le cahier des charges. En effet, bien que répondant au besoin, les routes initialement construites ne répondaient pas à une convention "fondamentale" de l'architecture REST : être intuitive, et allant du plus général au plus particulier. Nous avons ainsi changé cela pour obtenir les routes présentées plus haut, qui répondent à cette nécessité.

Enfin, un dernier élément qui diffère des prévisions initiales est la possibilité d'ajouter autant d'esclaves que l'on souhaite à un cluster, en connectant plusieurs bornes entre elles. Nous nous sommes en effet heurté à un problème causé par une des spécificités de l'I2C que nous ne connaissions pas avant d'y être confronté : le protocole distingue fortement le master du slave, et il est prévu pour établir une relation maître/esclave à sens unique (un maître défini et que des esclaves). Ainsi, un master ne peux pas devenir slave et inversement, ce qui dans notre cas, nous empêche de connecter deux clusters entre eux. Effectivement, deux bornes ne peuvent communiquer que en passant par des maîtres. Il faut donc qu'il y ai un maître sur la seconde borne pour communiquer, mais le protocole I2C ne veut également qu'un master et il faudrait donc repasser le maître en esclave après avoir effectuer la communication, ce qu'i2c ne permet pas. Nous n'avons malheureusement pas pu permettre cette fonctionnalité dans le rendu final, et il n'est ainsi possible pour l'instant que de réaliser une expérience avec un cluster de six Raspberries, ce qui offre tout de même de nombreuses possibilités d'utilisation. Cependant, le code est fait pour être réutilisable et évolutif en prévoyant déjà la possibilité d'avoir des stacks en plus (le broadcast est par exemple déjà implémenté avec cette possibilité d'un nombre plus grand de raspberries). Il serait possible d'écrire une classe MasterSlave qui pourrait à la fois servir de master pour la seconde borne, mais ne représenter qu'un slave pour le premier master et ne pas poser de problèmes. Une solution reste donc possible mais nous n'avons pas eu l'occasion de l'implémenter dans le cadre de notre PFA.

3.2 Le Daemon esclave et l'I2C

Dans ce module, nous pouvions distinguer deux aspects principaux. D'une part, le Daemon esclave, élément important permettant de contrôler les Raspberries et de récupérer des informations les concernant, codé en *C*. D'autre part, la communication I2C que les esclaves devaient utiliser pour communiquer avec le maître en recevant des ordres et en renvoyant une réponse.

3.2.1 Architecture

Le répertoire slave du dépôt contient le code C du Daemon. Le module *Daemon* (fichiers *daemon.c* et *daemon.h*) sert de programme principal en initialisant et en utilisant les autres modules. C'est en quelque sorte un Contrôleur, contenant la logique applicative gérant les divers cas d'utilisation et d'erreurs, mais servant également et partiellement de modèle, tout gérant les logs de fonctionnement dans un terminal. Les modules *I2Cslave* et *udpslave* implémentent les moyens de communication res-

pectivement en I2C et UDP. Ils proposent des fonctions d'initialisation, de connexion, d'envoi et de réception des données. Le module *commands* décrit les différentes commandes que le slave peut exécuter (demande de cpu, *shutdown*, *reboot*, demande d'adresse ip, demande d'adresse I2C), et expose des fonctions de découpage d'arguments, notamment pour "parser" le fichier *orders.txt* à la racine du dépôt ou pour découper les messages reçus en UDP, sous un format de chaînes de caractères séparées par des points-virgules.

Pour pouvoir surveiller à la fois la connexion réseau et le bus I2C, le Daemon slave utilise la fonction standard de C *select()*, qui permet d'enregistrer des descripteurs de fichiers (ici la socket UDP et le pointeur vers le *device* I2C) et de les interroger (*polling*) avec un *timeout* réglable. Ainsi, dès que des données sont disponibles sur un des descripteurs de fichiers ajoutés à l'ensemble surveillé (à l'aide de la macro FD_SET(fd, &read_fd_set)), un traitement adapté est effectué selon le descripteur de fichier repéré (test réalisé à l'aide de la macro FD_ISSET(fd)). La fonction *select()* renvoit le nombre de descripteurs de fichiers modifiés ou -1 si une erreur s'est produite.

3.2.2 Communication réseau : UDP

Le Daemon gère très classiquement une socket UDP à l'aide de l'API standard du C. Il maintient en mémoire deux couples (adresse IP, port) : le premier correspondant au master et le deuxième correspondant au slave. Lors de l'initialisation, celui-ci contacte le master dont l'IP est fixe et connue, en envoyant le message "*configure*". Ceci permet au master de l'enregistrer comme nouveau slave dans son modèle de données, et d'acquitter cet enregistrement auprès du slave.

Les messages respectent le même codage que l'I2C définis par le fichier *orders.txt* à la racine du dépôt. Celui-ci se présente sous la forme d'une liste de commandes séparées par des points-virgules. L'UDP diffère de l'I2C sur le retour : comme le master n'établit pas de connexion et de session (par exemple avec le protocole TCP), il est plus compliqué d'élaborer une réponse venant de ce dernier. Deux choix s'offrent à nous : soit le slave envoit dans sa réponse un identifiant à propos de la requête reçue, soit on stocke les requêtes envoyées en attente dans le master dans une quelconque structure de données qui sera maintenue au cours de l'exécution. C'est-à-dire qu'à l'envoi d'une demande à un slave il sera ajouté le retour souhaité et un timeout à cet ensemble de requêtes, puis une fois ce retour réceptionné la requête sera supprimée de la structure. Nous avons choisi le premier choix car beaucoup plus simple à implémenter et, après coup, tout à fait fonctionnel. Ainsi, chaque retour du slave a un codage précis en UDP, par exemple il sera envoyé en tant que réponse pour toute demande d'occupation CPU "cpu ;5.56" au master. Ce dernier détermine alors le slave qui l'a envoyé à l'aide de l'adresse IP de l'émetteur.

3.2.3 Fonctionnement de l'I2C

L'I2C est un protocole de communication permettant à un master de communiquer avec plusieurs slaves. C'est une communication qui permet d'envoyer des données octet par octet (un seul à la fois).

Un bus I2C est composée de deux câbles, le *SDA* (Serial Data Line [2]) et le *SCL* (Serial Clock Line [3]). Afin de communiquer, le master doit connaître l'adresse du slave. Cette adresse est composée de 7 bits. Le dernier bit envoyé permet de savoir si le master souhaite réaliser une lecture ou une écriture. En effet c'est toujours le master qui initialise la communication, le slave lui n'a pas le choix.

Quand le master veut envoyer un message à un slave, il lui suffit de vérifier que le fil SCL est libre. Si celui-ci l'est, il écrit ses données sur la ligne SDA et ensuite active la ligne SCL pour indiquer que le fil SDA contient des données. Le master attend que le slave reçoive ces données pour ensuite désactiver la ligne SCL afin d'éviter que le slave prenne encore en compte les données sur le SDA. Pour que le master sache que les données sont bien arrivées au niveau du slave (donc tous les 8 bits), il laisse la ligne

2. ligne de données bidirectionnelle
3. ligne d'horloge de synchronisation bidirectionnelle

SDA inactive afin que le slave l'active s'il a bien reçu les données. C'est le principe de l'acquittement. Si le slave ne répond pas alors le master doit retransmettre les données.

Pour que le master choisisse le composant auquel il souhaite envoyer des données, il doit envoyer "0" simultanément sur les lignes SDA et SCL pour signifier qu'il commence l'envoi d'un message. Puis, il envoie les 7 bits d'adresse ainsi que le bit d'écriture. Il attend ensuite qu'un des slaves (s'étant reconnu) acquitte, puis envoie les 8 bits de données au slave et finalement, il se met dans l'attente de l'acquittement et reprend son envoi de données. Les autres slaves savent que les données ne les concernent pas, et ils ne prêtent donc aucune attention aux données transférées sur la ligne SDA. Pour terminer la discussion, le master envoie en même temps sur la ligne SDA et SCL "1" afin de signifier la fin d'une communication.

Enfin si le master souhaite que le slave lui communique des données, il initialise une connexion, envoie l'adresse du slave dont il veut récupérer les données (avec le bit lecture), attend l'acquittement et enfin commence à lire un octet. S'il souhaite que le slave continue à parler, alors il réalise un acquittement et lit les 8 bits suivants, sinon il réalise un non-acquittement et termine la connexion.

3.2.4 Une communication I2C complexifiée

Nous avions dès l'établissement du document de spécifications remarqué que la communication au travers de l'I2C ne serait pas la partie la plus aisée de ce projet.

En effet, la première difficulté fut d'ordre matériel. Utiliser le module qui nous a été fourni afin de faire passer le Raspberry slave en mode esclave a posé un certain nombre de complications. La communication I2C est une communication qui a un fonctionnement maître/esclave. Le maître doit donner l'autorisation à l'esclave pour parler. Le problème est que les Raspberries ne sont initialement conçus que pour être des maîtres. Le Client nous a donc fourni un *overlay* afin de construire le module permettant à un Raspberry de devenir slave et d'avoir une adresse. Le problème est que la version du noyau ayant changée entre temps, une des fonctions n'était plus implémentée de la même façon nous avons donc du prendre rendez-vous avec le client afin de modifier cet overlay et que celui-ci fonctionne sur les Raspberries.

Le protocole I2C fonctionne de la manière suivante : le master envoie un message et attend une confirmation du slave. Si celui-ci souhaite envoyer un message, il doit alors attendre que le master l'y autorise. Nous avions déjà rencontré ce problème lors de la rédaction des spécifications fonctionnelles, et en avions conclu que cette manière de communiquer ne nous permettait que d'envoyer des données octet par octet. De plus, cette communication n'est que peu fiable : lors de tests nous avons pu observer que seuls deux messages sur trois ne renvoyaient pas d'erreurs. Il faut donc que le master "parle" tant que le message n'a pas été reçu par le slave. Pour cela, une gestion des exceptions a été mise en place côté master.

3.2.5 Écarts sur les prévisions

Dans le document de spécifications nous avions défini un protocole représenté par la figure 13.

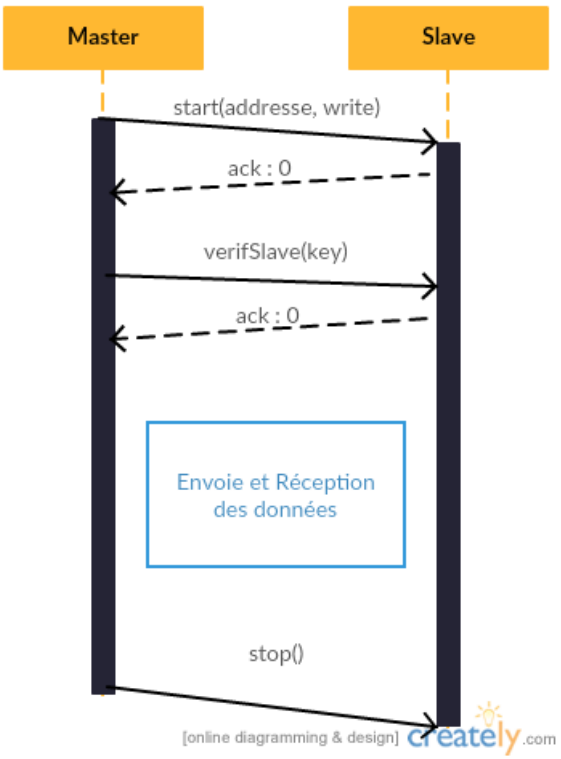

FIGURE 13 – Initialisation de la connexion en I2C

Nous souhaitions réaliser une gestion de connexions permettant tout d'abord de se connecter à un Raspberry sélectionné, puis d'échanger avec lui des données. Cela n'a pas été possible du fait de la connexion étant réinitialisée à chaque échange de messages, ceci étant causé par la bibliothèque Python SMBUS.

Nous avions aussi décidé de réaliser un *Tuple*[4] d'identification permettant de vérifier que les Raspberries auxquels nous parlions étaient ceux attendus. Cela a été compromis par la gestion de connexions. En effet cette dernière se réinitialise à chaque nouvel envoi de données. Nous pouvons donc supposer être sur le bon Raspberry lors de la demande de la clé, puis être sur un tout autre lors de l'envoi du message. Nous risquons donc d'envoyer un message dans le vide.

Nous avons ainsi réalisé le protocole ci-dessous.

4. Triplet de données

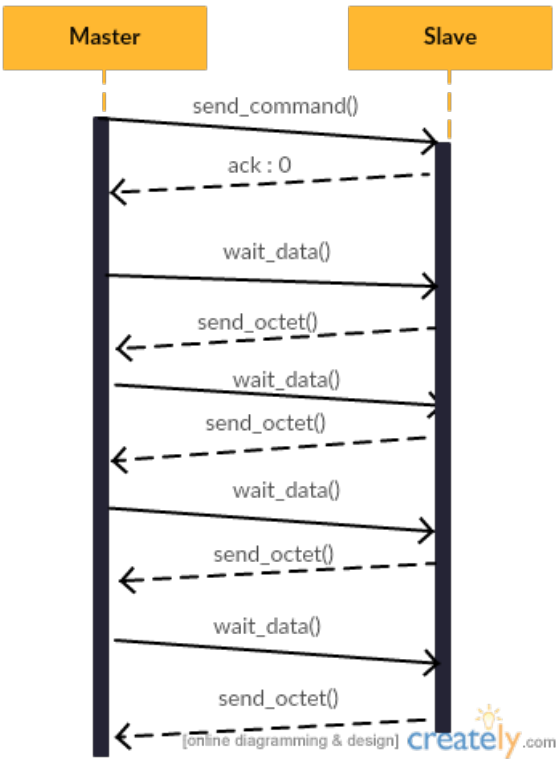

FIGURE 14 – Nouveau protocole I2C

Les messages envoyés entre le master et le slave ne dépassent pas quatre octets. En effet, la connexion devant être refaite à chaque octet, nous limitons à la fois les demandes et les retours.

Le nouveau protocole fonctionne donc ainsi : le master envoie la commande qu'il souhaite réaliser sur le slave grâce à un numéro. Il récupère ce numéro depuis un fichier texte "formaté", contenant l'ensemble des commandes implémentées sur le slave. Si l'envoi de l'instruction se déroule convenablement, le master donnera la parole au slave et se mettra en attente des quatres octets de réponse.

Concernant l'implémentation, le master possède une instance de la classe *Slave(PiDevice)* pour chacun des slaves auxquels il souhaite communiquer. Ainsi, l'envoi se faisant par l'implémentation d'un communicator présenté plus tôt : *CommunicatorI2C*, ce dernier possède une fonction *send_instruction* prenant tout simplement en paramètre une instance de *Slave*, un numéro correspondant à l'instruction et pour finir le nom associé à la commande, celui-ci permettant au slave de savoir s'il doit attendre une valeur de retour. En effet, afin de ne pas attendre désespérément une réponse à la suite d'une instruction sans valeur de retour sur le bus (shutdown par exemple), la classe *Slave* possède un ensemble statique *no_return_instructions* comprenant les noms des instructions ne nécessitant pas de read de la part du master. Par la suite, toute implémentation d'instructions sans retour sur le bus devra s'accompagner de la mise à jour de l'ensemble.

3.2.6 Implémentation

Afin d'être en accord avec ce protocole nous n'avons pas pu implémenter toutes les fonctions possibles sur un Raspberry comme nous le souhaitions. Nous avons donc dû faire des choix. Par exemple la commande *ls*[5] n'est dans notre cas pas une fonction à implémenter en priorité : elle retourne beaucoup de texte et n'est pas primordiale dans les tests réseau. Nous avons donc décidé d'implémenter les fonctions suivantes :

— **Allumer un Raspberry :** Le matériel nous le permet via la commande

```
sudo I2Cset 0x42 0
```

qui permet au niveau du Raspberry ayant cette adresse de mettre en route le bus I2C correspondant, et par conséquent d'allumer le Raspberry. Il s'agit d'une fonctionnalité déjà implémentée

— **Éteindre un Raspberry :** À la réception de la commande correspondante, le Raspberry envoie une réponse signalant son extinction proche, puis lance

```
sudo shutdown -h now
```

qui provoque donc son arrêt. Pour le rallumer il faudra d'abord éteindre le bus I2C avec :

```
sudo I2Cset 0x42 0xff
```

puis ensuite le (re)démarrer avec la même commande comme vu précédemment :

```
sudo I2Cset 0x42 0
```

Il nous a semblé utile d'avoir cette fonction en cas de problème afin de pouvoir éteindre le Raspberry.

— **Redémarrer un Raspberry :** cette fonction consiste en la même commande que précédement. Il suffit simplement de rajouter `-r` afin que le Raspberry s'allume à nouveau.

— **Afficher la charge CPU :** cette fonction retourne un pourcentage du CPU utilisé. Elle nous a paru utile également car elle permet de voir les statistiques du Raspberry mais aussi si le Raspberry est occupé ou si on peut lancer une tâche via UDP sur celui-ci.

— **Demande de l'adresse I2C :** l'adresse I2C est stockée en local dans les fichiers de configuration du Raspberry. Cette fonction pourra notamment servir pour les tests et vérifier que l'on parle bien au Raspberry auquel on souhaite parler.

— **Afficher l'adresse IP :** pour cela nous récupérons la sortie d'un ifconfig et nous donnons l'adresse IP utilisée dans la cadre de eth0. Pour retourner cette adresse, on retourne chaque entier de la notation humaine sur un octet. Il nous a semblé intéressant d'implémenter cette fonction sur l'I2C car le cluster va permettre de réaliser des tests sur les connexions réseau. Il peut donc être pertinent de récupérer ces adresses par l'I2C en cas de problème. De plus une adresse IPv4 peut être envoyée sur 4 octets en retournant 4 caractères.

— **Vérifier qu'un slave est connecté à Internet :** pour cela on analyse la sortie d'un ping pour vérifier que la connexion est bien faite. Cette fonction est aussi primordiale dans le cadre de tests réseau.

Afin que le slave sache quelles fonctions utiliser, un fichier texte indique la correspondance entre chaque fonction et l'octet à envoyer au slave. Dans le but de pouvoir communiquer avec le master, le slave a continuellement un daemon qui tourne grâce à un WHILE(1) [6]. Dès que la fonction détecte l'écriture d'un octet sur l'entrée I2C, celle ci est traitée. Si la demande est une demande de redémarrage

5. Simple commande listant le contenu d'un répertoire
6. Boucle infinie

ou d'extinction alors le Raspberry ne renvoie pas de réponse et s'éteint simplement. Pour les autres fonctions comme la demande de charge du CPU et la demande de l'adresse IP, le Raspberry va d'abord réaliser la fonction afin d'avoir les informations pour ensuite les envoyer.

Nous avons également une fonction test qui fait un simple ECHO afin de vérifier simplement la connexion I2C.

3.3 L'interface graphique

Le projet devait intégrer une interface graphique utilisant l'API REST du master afin de gérer visuellement le système. La conception de l'interface s'est orientée naturellement vers une solution simple mais efficace : des pages web utilisant bootstrap pour le design.

3.3.1 L'interface Web finale

L'objectif de cette interface est de proposer une vue d'ensemble du système, d'accéder aux détails de chaque Raspberries, et de fournir les fonctions proposées par l'API REST. Pour cela, nous avons découpé l'interface en trois vues. La première vue est une vue générale, on y trouve l'accès aux informations sur les Stacks ainsi qu'une liste des Raspberries existants dans chaque Stack. Ensuite il y a la vue d'une seule Stack, dans cette vue l'accès aux informations des Raspberries est plus complémentaire. Enfin la dernière vue concerne un Raspberry précis.

L'apparence de l'interface a été conçue à l'aide du framework Bootstrap. Lors du chargement d'une page, la structure de la vue est chargée ainsi que les scripts JavaScript. Les données formant le contenu de la page sont obtenues dynamiquement par des requêtes AJAX régulières (toutes les cinq secondes) à l'API REST qui les formatte en JSON.

L'ensemble des requêtes GET de l'API (qui permettent d'obtenir l'état du système) est accessible visuellement et réparti sur les différentes vues. La température n'est accessible que depuis la vue générale et la vue d'une Stack, les pourcentages d'utilisation du CPU et de la RAM se trouvent sur la vue d'une Stack et la vue d'un Raspberry.

Les requêtes POST (actions sur les éléments du système) sont accessibles à l'aide de différents boutons. Lors d'un clic sur un bouton, une confirmation est demandée (par exemple pour éteindre un Raspberry) avant que la requête ne soit effectuée.

Le bouton ON/OFF concernant le Raspberry fait appel aux routes /Rasp/id/stop et /Rasp/id/start respectivement. Il y a aussi un bouton Restart pour redémarrer un Raspberry déjà allumé (il fait appel à /Rasp/id/restart). Le bouton ON/OFF d'une Stack ne fonctionne que quand il est en position ON (il permet donc d'éteindre la Stack). Il est cependant possible de faire une nouvelle route pour allumer une Stack dans une prochaine version.

3.3.2 Les différences avec le cahier des charges

La principale différence que l'on peut observer entre la version finale de l'interface et le cahier des charges est le visuel en lui-même. En effet, nous avions établi une maquette permettant de présenter l'affichage que pourrait avoir notre interface. Hors, bien que la fonction des différentes pages (vue générale sur le cluster, vue détaillée d'un Raspberry etc) n'ait pas changé, le visuel en lui-même s'est beaucoup écarté de celui d'origine, en offrant une interface plus épurée et simple.

Dans les paragraphes suivants, la vue générale et la vue d'un Raspberry sera détaillé avec un rendu visuel. Dans un premier temps, voici un aperçu de la vue générale :

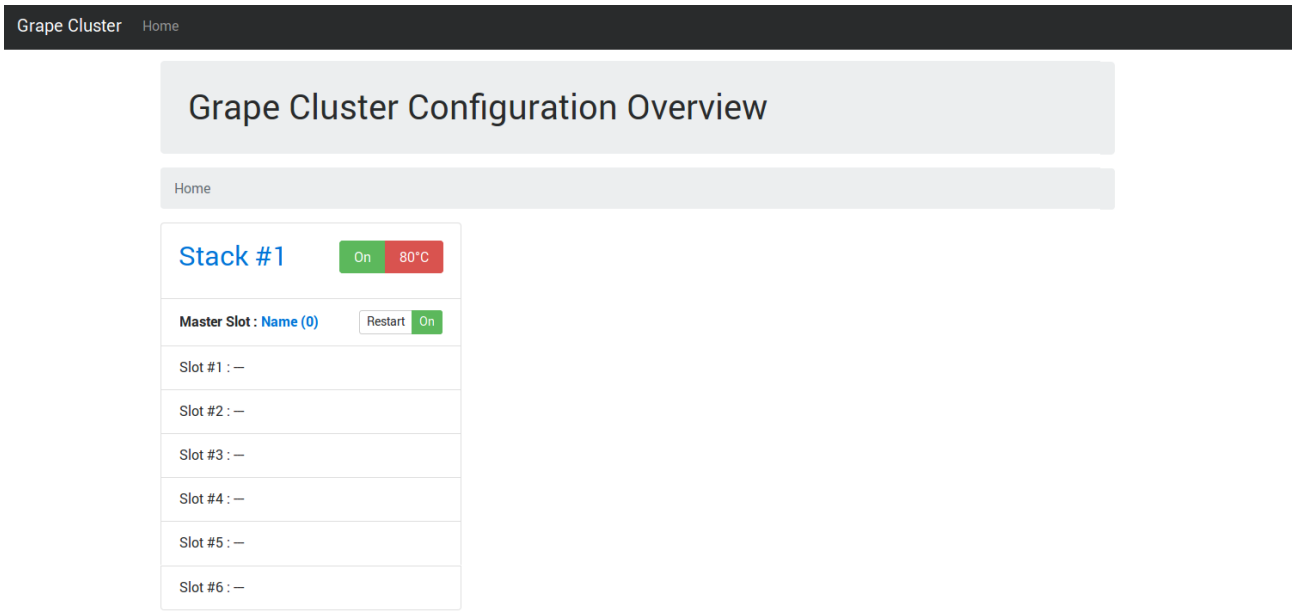

FIGURE 15 – Vue générale

Dans le cas présent, une seule *Stack* est détectée, il n'y a donc qu'une seule colonne correspondant à la Stack allumée. Dans le cas où plusieurs Stacks sont allumées, différentes colonnes sont ajoutées pour afficher toutes les Stacks. Une Stack n'est composée au plus que d'un Raspberry qui occupe la fonction de master et de six Raspberries slaves. Dans cet exemple, seul un master est présent. Il y a donc sur l'emplacement prévu à cet effet son nom et son identifiant apparents qui permettent d'aller directement sur sa vue spécifique. Les boutons restart et ON permettent de redémarrer ou éteindre le Raspberry. Enfin, le bouton ON à côté du nom de la Stack permet d'éteindre toute la Stack. À proximité de ce bouton se trouve la température donnée par l'API REST.

Si on clique sur le lien pour aller sur la vue du master, on obtient le rendu suivant :

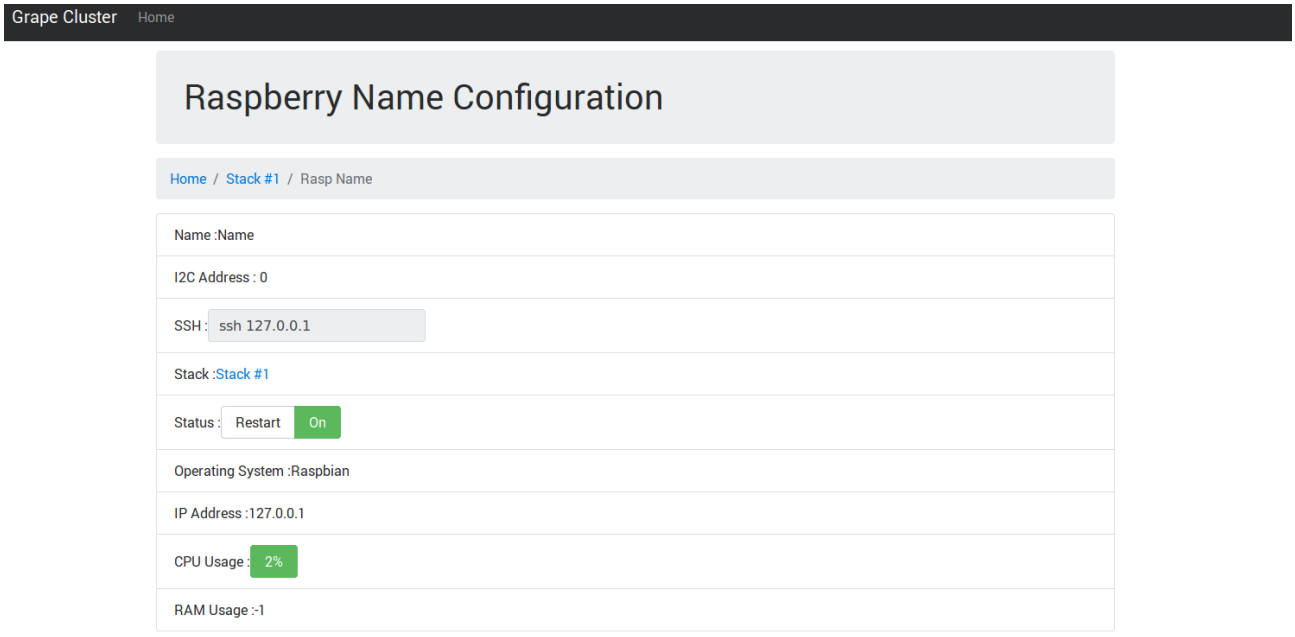

FIGURE 16 – Vue d'un Raspberry

Dans cet exemple, on constate que les informations sur le Raspberry sont plus complètes. En plus des informations disponibles sur la vue générale, l'addresse IP du Raspberry est donnée. La Stack contenant ce Raspberry est précisée et un lien permet de revenir sur la vue correspondant à cette Stack. On trouve également les informations relatives à l'utilisation du CPU et de la RAM. Ici seules les informations du CPU sont obtenues de l'API REST. Enfin le système d'exploitation installé est précisé et une commande pour se connecter directement au Raspberry est donnée.

L'interface finale ne permet pour le moment pas l'accès à un terminal pour lancer des commandes manuellement depuis ce dernier comme spécifié dans le cahier des charges. Cette partie n'a pas pu être implémentée, il faut donc se connecter en *ssh* sur les Raspberries pour lancer des commandes personnalisées. Une commande est rappelée à cet effet sur la page correspondant au raspberry qui doit être connecté.

3.4 Le script de l'image minimale

3.4.1 Un script modulable

Dans le but d'avoir une certaine modularité, nous avons donc mis en place un script permettant de créer à partir (par exemple) d'une image d'un système comme Raspbian, une image qui nous conviendrait à nous, dans le but de réduire le plus possible les actions superficielles et répétitives. En effet, nous avons besoin d'"augmenter" l'image basique du système Raspbian : ne serait-ce que pour pré-installer des paquets ou encore ajouter des modules au kernel. Notre script se divise en trois parties que nous allons vous exposer par la suite. Nous allons prendre l'exemple d'exécution simple où nous souhaitons simplement installer les paquets basiques, recompiler le kernel et installer le Daemon.

La figure ci-dessous permet de résumer dans son ensemble le scénario d'utilisation que nous allons voir. Les principales étapes illustrées par ce schéma seront repérables le long des explications qui suivent car mises en avant en **gras**.

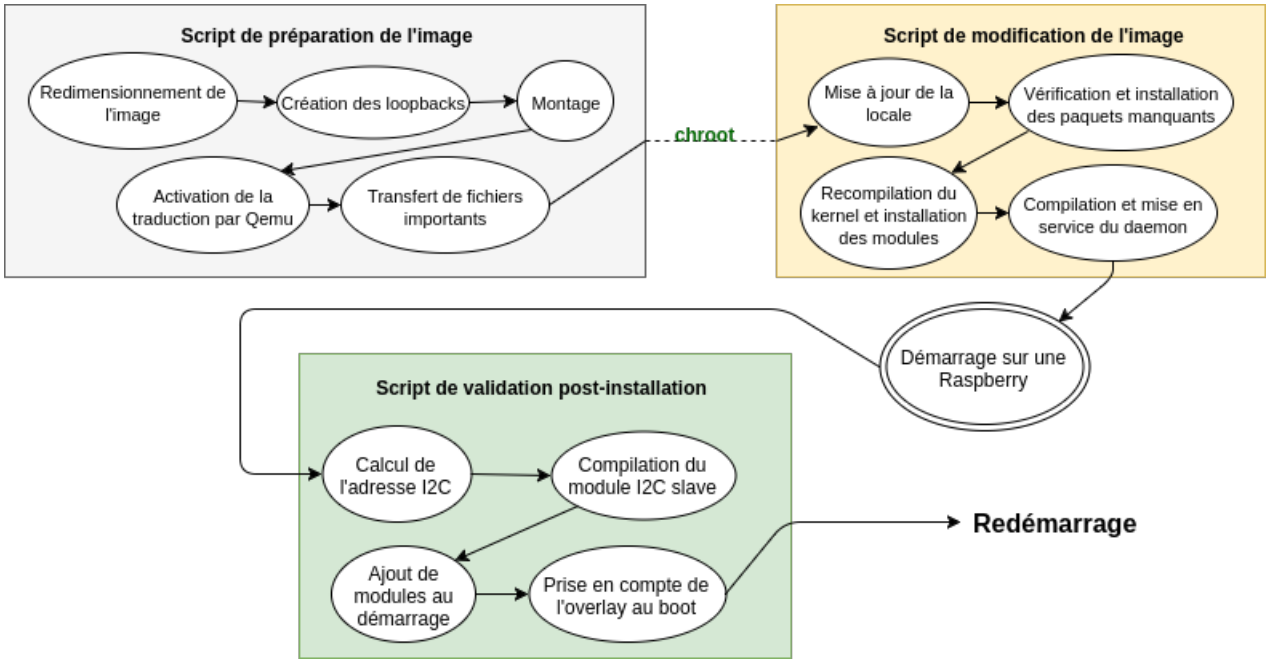

FIGURE 17 – Enchaînement d'actions du script

3.4.2 Management d'une image ISO

En effet, nous avons d'abord un premier script qui prépare l'image en tant que telle : le montage du système de fichier mais aussi (et surtout) son redimensionnement et repartitionnement. Dans cette

partie et celles qui vont suivre, nous allons parler d'utilisation de commandes Linux. Elles seront indiquées en *italique*.

- Tout d'abord nous avons la possibilité d'**agrandir l'image** du système Raspbian. En effet si nous souhaitons recompiler le kernel Raspbian il faudra rajouter au moins 1 Giga octet car les sources nécessaires venant de Git pèsent ce poids. Pour faire ceci, nous utilisons la commande *truncate* servant à étendre la taille du fichier que l'on peut lui donner en entrée.
- Ensuite, nous utilisons *losetup* dans le but d'utiliser les ***loop devices*** (fichiers spéciaux servant de référence à un système, au travers d'une image système) pour manipuler notre image ISO. *losetup* permet ainsi de traiter cette dernière dans son intégralité, comme un système à part entière. A l'aide de *fdisk*, nous supprimons puis recréons la partition principale de l'image Raspbian en augmentant sa taille. Finalement après avoir réalisé une vérification du système de fichiers nous utilisons *resize2fs* pour agrandir véritablement la taille de la partition en question.
- Puis vient le moment de **monter le système** (ici Raspbian). Ce dernier est un peu spécial puisque l'image ISO est divisée en deux partitions. Nous faisons donc le lien avec les *loop devices* grâce à *losetup* comme précédemment, et enfin nous montons les deux partitions (avec la commande *mount*) dans un espace de travail situé à l'emplacement du script. Nous sauvegardons de plus la dernière image montée (donc celle en cours de traitement). Afin de pouvoir continuer l'exécution du script, nous copions le script "in-*chroot*" dans le système de fichiers monté.
- Afin de réaliser le changement de racine (*chroot*) nous devons tout d'abord nous assurer de **la bonne traduction des données**. En effet Raspbian (par exemple) est prévu pour fonctionner avec un processeur de type ARM. Il nous est donc normalement impossible de communiquer directement avec le système puisque (sauf erreur) nous n'allons pas manipuler l'image depuis une machine ayant un ARM. Nous utilisons donc **Qemu** pour retranscrire l'exécution des programmes vers notre architecture. Nous lançons donc le *chroot* sur un exécutable Qemu (après avoir activé la traduction vers ARM) à qui nous faisons exécuter le *bash* du système Raspbian.
- Le démontage se fait naturellement avec *umount*, et libère les *loop devices* que nous avions utilisés. Il ne reste donc plus qu'à *flasher* l'image sur une carte SD.

Une fonctionnalité de cross-compilation du kernel a été ajoutée. En effet en l'utilisant (et bien que n'était pas totalement testée), plutôt que de compiler le kernel pendant le chroot et donc de perdre en performances, nous pouvons le réaliser directement en utilisant un compilateur spécial. Une fois cette compilation réalisée, le script n'installe pas les modules, mais se charge simplement de les copier dans le Raspberry. Si l'utilisateur souhaite les installer, il faudra qu'il utilise le troisième script qui se chargera de le faire. Nous ne réalisons pas l'installation car ce sont des fichiers d'une nature assez spéciale, puisque compilés pour un certain système de fichiers. Il faut que ce soit à la Raspberry de les installer, sinon lors de la copie de ces fichiers sur une carte SD, ces derniers sont totalement faussés et l'image devient inutilisable.

3.4.3 Préparation d'une image Raspbian

Le *chroot* dans l'image se fait à l'aide de Qemu, du *bash* de l'image mais également d'un script qui va jouer le rôle d'intermédiaire, et qui est donné à exécuter au *bash* en question. Ce dernier s'axe lui également sur plusieurs points.

- Tout d'abord afin d'éviter la multiplication de messages d'erreurs désagréables tout au long du script, nous **mettons à jour la *locale*** afin d'avoir la certitude d'avoir la bonne configuration de langage.

- Nous réalisons par la suite une **vérification de tous les packages nécessaires** pour le bon fonctionnement du script, et du Raspberry pour son utilisation future. Si un package n'est pas présent, **nous l'installons simplement**.

- Pour pouvoir utiliser l'I2C convenablement, il nous a fallu **recompiler partiellement le kernel Raspberry**, c'est à dire les modules liés à la communication pour l'I2C. Le script réalise donc

un *clone* du dépôt Git du kernel Linux en question, puis lance toute une série de compilation de modules et éléments importants dont nous avons besoin, ainsi que leur installation. Après cela, nous copions les fichiers produits et notamment les *Device Tree overlays* : ce sont des fichiers qui permettent de supporter plusieurs configurations matérielles avec un seul kernel, sans de manière explicite charger ou *blacklister* des modules kernel.

— Après cela nous récupérons un nouveau dépôt Git, contenant lui les exécutables plus particuliers pour l'I2C modifiés par nos soins, ainsi que les **sources du Daemon**. Nous produisons les fichiers de données adéquats et **l'exécutable** via plusieurs étapes de compilation.

— Finalement, à l'aide d'un script préconçu, nous plaçons l'exécutable du Daemon en tant que **service à lancer au démarrage**. Ce dernier permet d'utiliser donc le Daemon comme un service : un typique appel à service permet de gérer l'état d'exécution de ce dernier (`service start Daemon` permet par exemple de lancer le service en question). De plus le script utilise le **logger** pour faire des logs dans le */var/log/messages* afin de tenir au courant d'éventuels problèmes. Ce script est simple, et facilement extensible.

— Si l'option est précisée, nous réalisons une mise à jour complète du système grâce à son manageur de paquets.

— Enfin, nous activons le **ssh** vers le système (en ajoutant un simple fichier nommé "ssh" dans le répertoire */boot*.

Toutes les actions menées par ce script sont optionnelles. Par exemple, il est totalement possible de lancer le script principal en précisant l'option "*-chroot-option=chroot-only*" pour atterrir directement en *chroot* sur le système de fichiers sans que le script ne se soit exécuté. Nos deux scripts se nomment

donc respectivement *armmanager* et *Rasparchitect*. Le développement et explications que nous venons de faire pourrait correspondre à la commande de lancement du script suivante :

```
./armmanager.sh -m image.iso -r -c -co=upgrade-clean
```

L'option *m* sert à indiquer que nous voulons monter l'image ISO (ou IMG), l'option *r* montre que nous voulons d'abord l'agrandir (par défaut d'1 Gigaoctet). L'option *c* demande qu'une fois l'image montée, nous réalisions un *chroot* à l'intérieur de celle-ci (tout chroot réalisé à l'intérieur se fait d'abord par le lancement du script "in-chroot" *Rasparchitect*). Nous rajoutons finalement des options de *chroot* avec l'option *co* et l'argument *upgrade-clean*, signifiant que nous souhaitons réaliser une fois le script *Rasparchitect* terminé, une mise à jour ainsi qu'un nettoyage de tous les paquets du système. D'autres options de chroot existent, comme par exemple le *install-only*, ou le *chroot-only*. Ces options permettent de choisir quelles actions faire ou ne pas faire une fois le chroot lancé.

3.4.4 Script post-installation

La totalité des installation ne peut se faire en *chroot* et nous avons donc besoin d'exécuter un nouveau script qui se chargera de terminer totalement les installations.

En effet se dernier se charge de calculer en fonction de deux paramètres la valeur de **l'adresse de l'esclave** en hexadécimal : le numéro de la *stack* sur laquelle est présente le raspberry et sa position sur cette dernière. Suite à cela nous **recompilons le module kernel I2C slave** avec cette adresse, et créons dans le répertoire */etc/modprobe.d* un fichier **i2c.conf** permettant au module, lorsqu'il se charge, de bien s'initialiser avec la bonne adresse I2C. Pour finir, nous appliquons **l'overlay** au Raspberry afin que ce dernier le chargement au moment du démarrage, dans son */boot/config.txt*. Un simple redémarrage, et l'esclave est enfin totalement prêt !

3.4.5 Les différences avec le cahier des charges

D'après le cahier des charges, les objectifs du script semblaient être assez simples. Cependant nous nous sommes heurtés à un nombre assez conséquent de problèmes. Tout d'abord le choix du langage avec lequel écrire ce script : le bash semblait être une "évidence", cependant nous sommes en premier partis sur un script python. Finalement nous avons terminé en le faisant entièrement en bash car ce dernier est exclusivement composé d'appel de commandes unix et de manipulation de fichiers d'images disques.

Nous avions souhaité demander le moins possible l'intervention de l'utilisateur pour lui éviter d'avoir à entrer le mot de passe administrateur trop de fois. Le package nommé monter un système de fichiers sans droits d'utilisateurs en utilisant un système de type FUSE. L'idée de manipuler l'image sous cette forme pouvait être intéressante, cependant le changement de racine induit par le *chroot* était apparemment impossible avec ce système de fichiers (en rajoutant à cela un trop grand nombre de paquets et de manipulations nécessaires simplement pour monter l'image ...). Nous avons donc opté pour demander périodiquement en précisant pour quelles commandes à l'utilisateur, d'entrer son mot de passe.

Le script respecte le plus possible le cahier des charges initial, et permet une importante modularité. Il ne fait pas simplement qu'installer des paquets et préparer le système, il permet également de fournir à l'utilisateur une micro-plateforme lui permettant, s'il le souhaite, de travailler lui même en *chroot* sur le système Raspbian. Via un certain nombre d'options il peut de plus personnaliser l'installation qu'il souhaite faire, pour ainsi s'adapter au mieux à ce qu'il souhaite.

3.5 L'image minimale et le TFTP

Dans l'idée, il était souhaité que chaque Raspberry au démarrage mette en place une vérification de son système ou de l'image qu'il possède, et se mette éventuellement à jour si son image n'est pas la dernière en date. Ceci est rendu possible par l'utilisation de plusieurs choses que nous allons exposer. Un schéma explicatif ci-après est proposé pour éclaircir l'idée.

— L'utilisation d'une **image minimale** : *Tiny Core Linux* est un système dont le but principal est de fournir un système léger et rapidement opérationnel dans l'optique de réaliser des traitements applicatifs simples. La carte SD d'un Raspberry est donc partitionnée en trois : la première sert à Tiny Core en tant que tel, la deuxième laisse un petit espace de stockage à Tiny Core dans lequel nous placerons des données importantes, et la troisième supposée "vide" sert à accueillir la nouvelle image qui sera "*flashée*" à la volée.

— La mise en place d'un **serveur TFTP** [7] côté maître : En effet au démarrage, chaque Raspberry esclave contacte le serveur du maître en lui demandant la vérification de l'image souhaitée. Pour cela un checksum [8] *md5* caractérisant l'image est généré. Le Raspberry démarrant compare le checksum qu'il possède avec le distant, et si différences il y a il récupère l'image distante pour l'importer directement sur son système. Une fois ceci fait, il redémarre en sélectionnant d'office le système nouvellement (ré)installé.

7. Trivial File Transfer Protocol
8. Somme de contrôle, "empreinte" d'une donnée

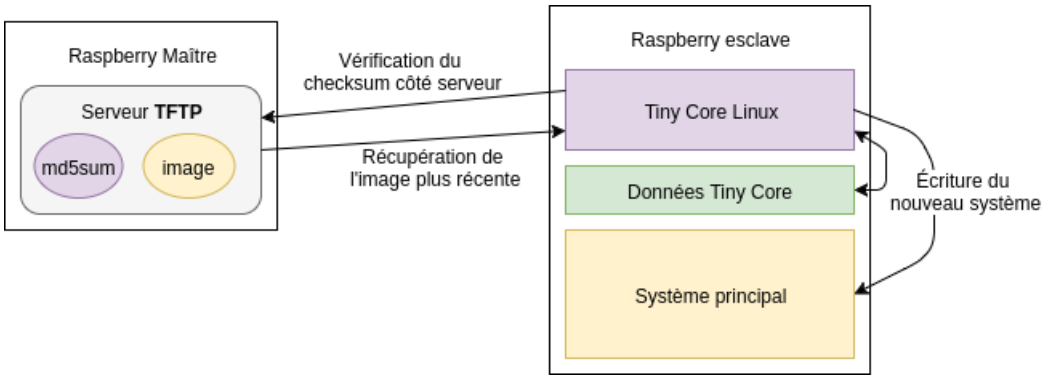

FIGURE 18 – Schéma des interactions entre Tiny Core et le TFTP du maître

3.5.1 Écarts de réalisation

Cette partie n'a malheureusement pas été totalement fonctionnelle et totalement testée. En effet, nous nous sommes inspirés de travaux de chercheurs du *SE Research Group* portant sur le "Grape : un cluster basé sur des noeuds de Raspberry Pi", et avons du nous les approprier notamment en adaptant les parties de vérification de l'image, et actions associées.

La difficulté s'est trouvée dans l'implémentation de ce dernier. En effet après des tests entre une Raspberry un serveur TFTP, le raspberry contacte bien le serveur mais du fait de la nature de ce dernier, il est incapable de transférer l'image disque en question. Un paramètre de type *BLK_SIZE*[9] stoppe le transfert indiquant qu'il ne peut faire transiter un fichier aussi important. Cela est apparemment un bug connu du serveur TFTP *atftpd*. Il faut donc passer par l'installation et la configuration d'un autre serveur TFTP, à savoir *tftpd*.

3.6 Bilan global : le projet dans son ensemble

Finalement, après avoir développé parallèlement chacun de ces modules séparément nous avons pu les connecter entre eux pour former le produit final. Cette étape, moment crucial du projet, a nécessité toute notre attention, notamment en matière de test.

3.6.1 La vérification : s'assurer de la pertinence de la solution

Lorsque l'on en vient à parler du résultat final du projet, il faut évidemment pouvoir estimer la qualité de celui-ci. Cette qualité correspond principalement, dans le cadre du PFA, à la bonne satisfaction des besoins du client. Il convient donc de s'assurer de cela à travers différents tests, pour s'assurer que le produit correspond aux attentes établies. Nous avons donc établie dans un premier temps différentes politiques de tests pour chacun des modules, afin de les tester séparément. La plupart de ces tests on été des tests d'intégration (tests testant une partie spécifique du projet), s'assurant donc qu'un aspect particulier fonctionnait bien. Nous avons pour cela utilisé des fonctions bouchons pour combler les dépendances entre les différents modules et pouvoir les tester séparément. Cette première partie des tests a donc été réalisée pendant le projet, au fur et à mesure de l'avancement. Dans un second temps, un des aspects les plus intéressant d'un point de vue vérification a été de réaliser des tests du projet dans son ensemble. En effet, c'est une fois lié et formant un projet uni et cohérent qu'il était important de tester les différentes fonctionnalités, et s'assurer du bon fonctionnement de la solution. Nous avons donc établie une politique de tests assez large, en remettant en place des tests d'intégration, sans utiliser cette fois-ci de bouchons. Cela nous a permis d'une part de s'assurer de la non-régression par rapport au moment où tout était séparé, et de s'assurer de la qualité du code. De plus, nous avons effectué une batterie de tests de validation, tests ayant pour but de s'assurer que

9. Block Size

toutes les exigences du client sont respectées. Pour cela, nous avons définit différents cas d'utilisation définissant des scénarios à tester, permettant de valider la correspondance avec le cahier des charges. Ces scénarios décrivent succinctement une série d'événement qui sont supposés se succéder suite à une action de l'utilisateur. Par exemple, un scénario concernant l'affichage des informations d'un Raspberry, qui se traduira par le clique de l'utilisateur sur l'interface web, qui va provoquer l'envoie d'un message au maître, qui demandera lui même à la Raspberry correspondante d'envoyer ses informations. On a ensuite l'envoi des messages retour de l'esclave au maître, puis de ce dernier à l'interface, et l'affichage. Ce court scénario décrit donc une succession d'événement attendue, que l'on va pouvoir comparer à la suite d'événement que l'on observera lors du test en condition réelle. C'est ainsi que les fonctionnalités ont pu être toutes testées une à une, et que l'on a pu s'assurer de la qualité et la pertinence du produit finale.

3.6.2 La fusion en elle-même

Avec toute cette politique de test définie, nous pouvions donc nous assurer que la fusion finale nous conduisait bien à un produit convenable. Nous avons pu donc effectuer cette étape avec tous les outils en main.

D'une part donc, nous avons connecté le Daemon esclave avec le Daemon du maître. Cette connexion paraissait aux premiers abords relativement aisée, la communication via les sockets fonctionnant de chaque côté, il semblait naturel que les deux ensembles communiqueraient automatiquement. Cependant, nous avons tout de même eu à faire à quelques barrières, notamment du côté matériel, pour faire communiquer correctement un Raspberry maître à un ou plusieurs Raspberry esclave. Des tests utilisateurs supplémentaires ont donc été réalisés pour s'assurer que la fusion n'entraînait pas de régression.

D'autre part, nous avons également réalisé la connexion entre le Daemon maître et l'interface Web. Cette partie, paraissant peut être moins évidente en premier lieu, s'est en fait révélée plutôt naturelle. Elle a en fait été réalisée de manière progressive, en remplaçant petit à petit les fonctions bouchons de l'interface pour les remplacer par les traitements liés aux routes REST définis en python. Du fait de l'architecture REST et de la bibliothèque Flask, il a été bien plus facile de lier les deux modules ensembles, et nous avons pu au final obtenir le résultat escompté rapidement.

Un autre aspect important de cette phase de connexion a été la fusion réelle sur le dépôt, pour obtenir un dépôt clair et structuré. D'un aspect purement structurel, nous avions déjà pris les devants en ordonnant correctement chacune de nos branches à travers des dossiers séparant maître, esclave et interface web. Le but était bien sûr de rendre le "merge" (fusion avec git) bien plus facile. Cependant, nous avions commencé à faire des tests de connexion entre différents modules, notamment entre le maître et l'interface, avant de réaliser correctement la fusion. Cet aspect progressif, bien qu'assez utile pour s'assurer du bon fonctionnement et de la bonne validité des tests, a posé ensuite problème lors de merge définitif, où nous avons du beaucoup plus effectuer certaines fusions (de fichiers ou de dossiers) à la main plutôt que de le faire faire automatiquement par l'outil de versionning. Nous avons ainsi finalement pu obtenir un dépôt cohérent et ordonné. Cela nous a ainsi permis d'apporter une bonne expérience d'un cas réel de fusion à plutôt grande échelle, avec qui plus est des erreurs qu'il a fallu pouvoir gérer. Ce dernier aspect du projet a donc également apporter sa part d'acquis et d'observation.

Conclusion

Dans le cadre de notre projet au fil de l'année, nous avons ainsi pu réaliser l'objectif premier qui était fixé : une interface graphique opérationnelle pour utiliser et interagir avec le grape cluster déjà conçu. Le résultat final obtenu peut bel et bien être utilisé pour communiquer à un master des requêtes et récupérer les informations pour les afficher sur l'interface, et ainsi permettre l'utilisation des grapes cluster lors de tp ou d'expérimentation.

Pour arriver à ce résultat, nous avons mené à bien toute la phase de conception qui aura permis de bien définir les enjeux de ce projet et la manière de le compléter. Nous avons également utilisé et développé des méthodes de gestion pendant la phase de développement en s'inspirant des méthodes agiles, pour travailler efficacement et conjointement, et résolu divers problèmes qui ont permis d'affiner notre jugement quant à la qualité du produit final.

De plus, nous avons développé nos quatre modules en parallèle : un Daemon pour le master ainsi qu'un autre pour les esclaves, mais aussi un script de création d'image pour automatiser et rendre plus modulable le processus et enfin une interface graphique pour commander tous ces modules. Ces derniers ont pu être reliés pour ne former qu'un même produit autonome et cohérent, tentant de répondre le plus possible au besoin.

Finalement, ce PFA nous aura permis d'acquérir beaucoup d'expérience en ce qui concerne la réalisation d'un projet de plus grande envergure que ce que l'on peut avoir l'habitude de traiter à travers les autres projets. De par à la fois la taille de l'équipe plus grande qui demande une organisation plus rigoureuse et efficace, mais aussi le travail à accomplir qui est plus conséquent. De plus, il nous aura confronté à plusieurs obstacles, que nous aurons réussi à surmonter en apprenant de ces situations. La satisfaction d'avoir mener à bien ce projet est d'autant plus grande que nous avons dû résoudre des problèmes totalement nouveaux pour nous, et que leur résolution à non seulement permis d'améliorer le produit final, mais également de nous enrichir nous-même.

Ainsi, ce produit conclut ce projet, mais ne reste pas pour autant un produit final qui n'évoluera plus. En effet, il reste encore plusieurs fonctionnalités ou aspects du cluster montrés intéressant que nous n'avons pas pu traiter dans le cadre de ce projet, ou qui restent encore à explorer. Par exemple, le contrôle de l'écran LCD ou de la molette qui n'a pas pu être inclu dans le produit final. Il peut aussi être envisagé que ce cluster soit ensuite adapté pour des expériences Big Data qui nécessitent de paralléliser les calculs. Cette interface déjà opérationnelle et efficace est donc probablement vouée à évoluer encore ou à être complétée par d'autres futurs modules, ce qui sera rendu plus facile par le caractère modulable que nous nous sommes attachés et efforcés de garantir dans le code de ce projet.

Annexes

Documents de Spécifications
-
Grape Cluster

Author :
Cyril Bos
Paul Breton
Benjamin Dos Santos
Nicolas Hannoyer
Marion Lafon
Jason Pindat
Nicolas Vidal

Encadrant :
David Renault

Client :
Floréal Morandat

7 avril 2017

Table des matières

1 Introduction

Cette partie introduit le contexte du projet et les différentes dénominations qui seront utilisées tout au long de ce cahier des charges.

1.1 Contexte et objectif du PFA

Le GrapeCluster permet de réaliser des TP et des expériences réseaux en utilisant des machines physiques réelles plutôt que des machines virtuelles (QEMU, VirtualBox, ...). Ainsi nous pouvons obtenir certains cas qui ne peuvent être émulés, par exemple une machine qui tombe subitement en panne. On obtiendra ainsi des erreurs habituellement invisibles lors de l'utilisation de machines virtuelles. Ce cluster est également nettement moins coûteux qu'un parc de machines réelles. Il est envisagé que ce cluster soit ensuite adapté pour des expériences Big Data qui nécessitent de paralléliser les calculs.

1.2 Présentation de l'existant

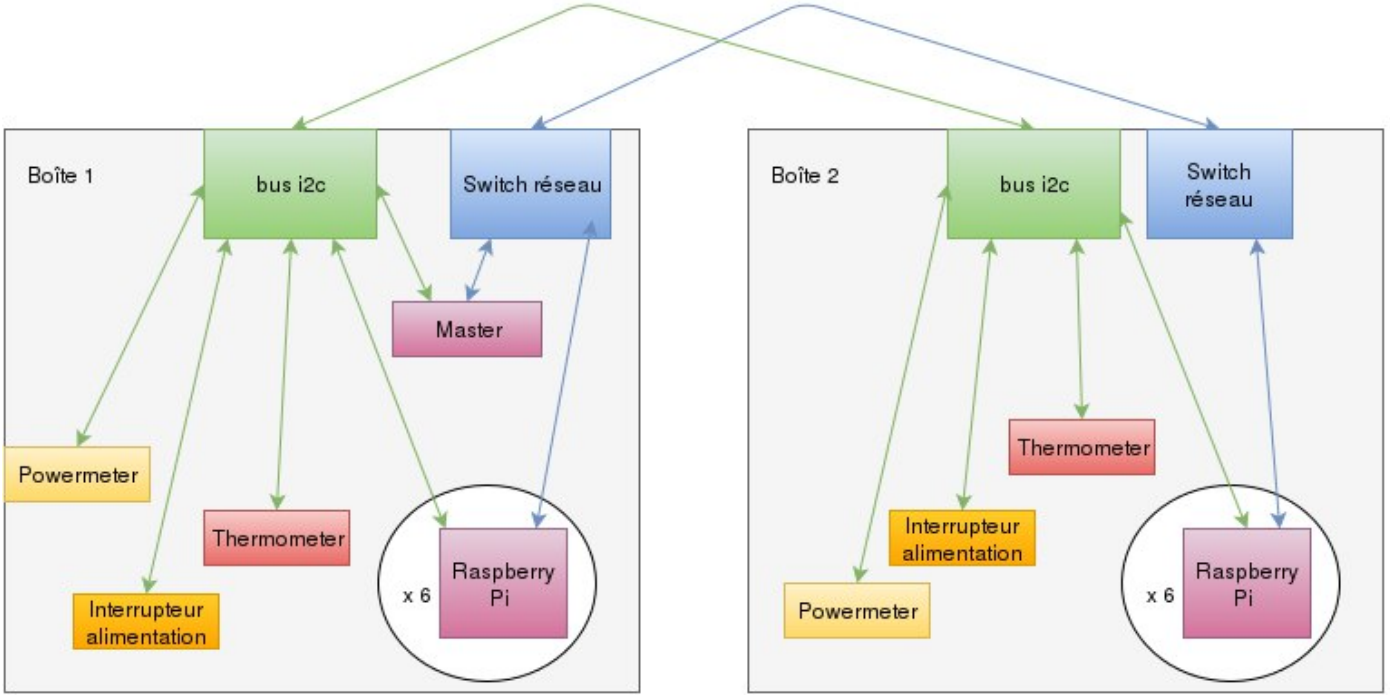

FIGURE 1 – Architecture matérielle

Le cluster est composé de stacks interconnectées par un bus i2c. Il est dirigé par une seule Raspberry Pi dite *master* contenue dans une des stacks. La borne contenant le *master* contient donc 6 autres Raspberries Pi dites *slaves*. Les autres stacks contiennent seulement 6 *slaves*. Chaque Raspberry Pi présente 3 ports i2c dont un est relié au bus i2c de sa borne. Les stacks communiquent ainsi grâce à leurs interconnexions i2c. Chaque borne contient également des sondes permettant de récupérer la température et l'utilisation électrique de chacune de ses Raspberries ou de la totalité d'une borne. Un autre composant électronique permet de couper l'alimentation d'une Raspberry souhaitée.

2 Besoins fonctionnels

Cette partie décrit l'ensemble des fonctionnalités attendues par le client.

2.1 Installation et organisation des slaves

2.1.1 Installation

A l'aide d'un script, il devra être possible de rendre générique l'installation de systèmes d'exploitation. Le script doit ainsi dans le but de configurer une image :

— Charger une image d'un système d'exploitation sur une machine (quelconque, mais ayant les outils nécessaires) dans le but de le configurer,
— Placer le daemon dans les processus à lancer au démarrage (le classer en tant que service, voir partie 4.2), et résoudre toutes les dépendances de paquets (notamment par exemple pour la communication i2c),
— Décharger l'image, la rendant prête à l'emploi

Une fois l'image disque obtenue, il suffira de disposer d'un script positionné et exécuté depuis l'image minimale d'un slave permettant de flasher cette dernière sur sa carte SD, puis de faire rebooter le slave afin de le faire démarrer dessus.

2.1.2 Partie slave

Chaque slave fera tourner en fond un processus nommé *Daemon* qui aura pour but de récupérer les informations lors de communications avec le master. Il devra ainsi :

— Arrêter, relancer la machine
— Échanger des données avec le master sur les ports i2c (messages courts) et UDP (messages plus importants en taille)
— Organiser la connexion au master (IP, ...)
— Établir des sauvegardes du système (données de configuration)

2.2 Contrôle des slaves par le master

Au niveau logiciel, un driver écrit en C est fourni pour faire communiquer une Raspberry avec son bus i2c. La liste des commandes à envoyer en i2c est :

— Allumer un slave
— Arrêter un slave
— Redémarrer un slave
— Demander le tuple d'identification d'un slave (voir partie 4.3)
— Demander le statut d'un slave

Le daemon du master doit permettre d'exécuter des ordres sur un ou plusieurs slaves en particulier.

— Parler à un slave via le protocole (envoyer des commandes : arrêt, changer d'image ...)
— Ouvrir un canal pour permettre au slave de parler
— Demander des informations d'état sur une stack
— Backup des informations de configuration des slaves (voir partie 4.3)
— Se connecter à des slaves et organiser le réseau des Raspberries (IP)

Les requêtes seront traitées de manière asynchrone.

2.3 Interface utilisateur et serveur Web

Afin de faciliter l'interaction avec l'utilisateur final, un serveur web doit pouvoir interagir avec le daemon du master pour utiliser les slaves.

— Interface avec un schéma interactif de l'ensemble du réseau GrapeCluster
— Possibilité de demander des informations d'état sur les stacks
— Sélectionner une ou plusieurs Raspberries pour installer de manière individuelle ou générale une image disque
— Sélectionner une ou plusieurs Raspberries pour leur envoyer des commandes dans un terminal (possibilité de réaliser un ssh sur ces machines, ftp ...)

3 Besoins non fonctionnels

Cette partie décrit l'ensemble des besoins permettant de mettre en oeuvre les besoins fonctionnels.

3.1 Qualité du code

L'un des principaux besoins non-fonctionnels qu'il faudra satisfaire est le besoin de refactoring facile. Nous allons ainsi utiliser des outils tels que Artistic Style pour formater le code, et les directives fournies par la norme PEP-008 concernant le code en Python. Ceci aura pour but de faciliter au maximum la possibilité de modifier le code ou le reprendre, en fournissant un code clair et propre, à l'aide de conventions de codage.

3.2 Mode de développement

Nous utiliserons un dépôt versionné qui sera un fork du dépôt contenant l'API de base. Nous réaliserons des pull-request qui permettront au client de revoir le code et valider les changements. Cette utilisation du dépôt permettra de toujours s'assurer de garantir la validité et la progression du code. En effet, les conditions de développement du projet vont mener à deux autres besoins non fonctionnels : le besoin de tests, qui devront couvrir les différents aspects du projet et devront pour cela s'appuyer sur différents scénarii, et le besoin d'amélioration incrémentale : chaque branche terminée devra garantir une progression et une non-régression par rapport à la précédente.

3.3 Automatisation des processus

Un important besoin fonctionnel sera aussi le caractère automatisable de certains processus. En effet, le client aura comme première utilisation des GrapeCluster-s de faire des tests réseaux qui auront la fréquente conséquence de remplacer un Raspberry par un autre, particulièrement pour cause de panne matérielle. De plus, ce projet est destiné à être repris ultérieurement, que ce soit par d'autres étudiants ou post-doctorants. Par conséquent, la plupart des processus de configuration et d'installation devront être automatisables.

3.4 Interface

En ce qui concerne l'interface utilisateur, il sera nécessaire d'utiliser Ajax. Il est important pour le client que les mises à jour de l'interface se fassent côté client, pour ne pas avoir à recharger la page à chaque action. En effet, on recherche un certain caractère optimal en ce qui concerne les transferts d'informations. Le but est de ne pas surcharger les communications. L'utilisation de cette technologie est donc essentielle.

3.5 Sécurité

L'aspect sécurité n'est pas une priorité en premier lieu selon l'avis du client, car le cluster fonctionne en réseau local coupé de tout autre réseau extérieur et l'accès à ce réseau se fera en branchant une machine à un switch d'une stack. L'accès physique au cluster sera donc réservé à un personnel restreint. On devra en revanche s'assurer que les Raspberries Pi de modèle 3B n'activent pas leur interface wifi.

3.6 API Rest

Selon les directives du client, les différentes informations et les ordres pourront être émis à l'aide de requêtes HTTP via une API REST dont les routes permettent l'utilisation du cluster sans forcément passer par l'IHM. Ainsi, il sera aisément possible d'écrire des scripts automatisés de configuration ou d'utilisation du cluster à l'aide d'outils comme `curl`. Les différentes actions possibles et les routes associées sont spécifiées dans la sous-section 4.6.

Bibliothèques fournissant un routeur web voire API Rest en Python :

- Falcon : micro framework très minimaliste et le plus performant. N'inclut pas de moteur de template, d'ORM. Sous license Apache 2.0.
- Bottle : micro framework semblable à Falcon. Inclut un système de templating, quelques utilitaires (upload de fichier, ...). Sous license MIT.
- Flask : micro framework plus complet et flexible. Possède une extension Flask-RESTful permettant d'étendre le protocole. Sous license BSD.

— Eve : framework de plus haut niveau basé sur Flask, nécessite une base de données MongoDB pour fonctionner. Moins flexible mais offre des fonctionnalités intéressantes comme la validation du format des données par la bibliothèque Cerberus et une gestion des évènements callback en utilisant la bibliothèque Events. Plus adaptée à une application web complète et plus classique qu'à un module qui sert principalement à acheminer les données. Sous license BSD.

Au vu de la méthode de développement très incrémentale, la bibliothèque retenue est Flask car elle permet une intégration d'évolutions plus aisée. Elle est flexible car modulaire et extensible, donc il est possible d'utiliser toutes ses fonctionnalités ou non suivant la nécessité de ses modules dans l'implémentation. C'est un framework mûr et sa documentation est très complète. La nécessité pour la bibliothèque Eve d'utiliser une base de données est contraignant et alourdirait la tâche de développement. L'IHM requiert un système de templating qui n'est pas disponible avec Falcon. Bottle intègre un système de routage semblable à Flask mais n'est pas aussi bien maintenu et n'est pas modulaire (un seul fichier, pas de système d'extension aisé).

Nous utiliserons une communication avec Ajax côté client qui permettra facilement de rendre dynamiques les pages de l'IHM, c'est à dire de rajouter une actualisation des données en "temps-réel". Et ainsi connaître au plus vite toutes les actions qui pourraient se passer sur les slaves.

4 Architecture et Conception

Cette partie présente en détail la solution logicielle qui sera premièrement développée, ainsi que son fonctionnement.

4.1 Architecture logicielle globale

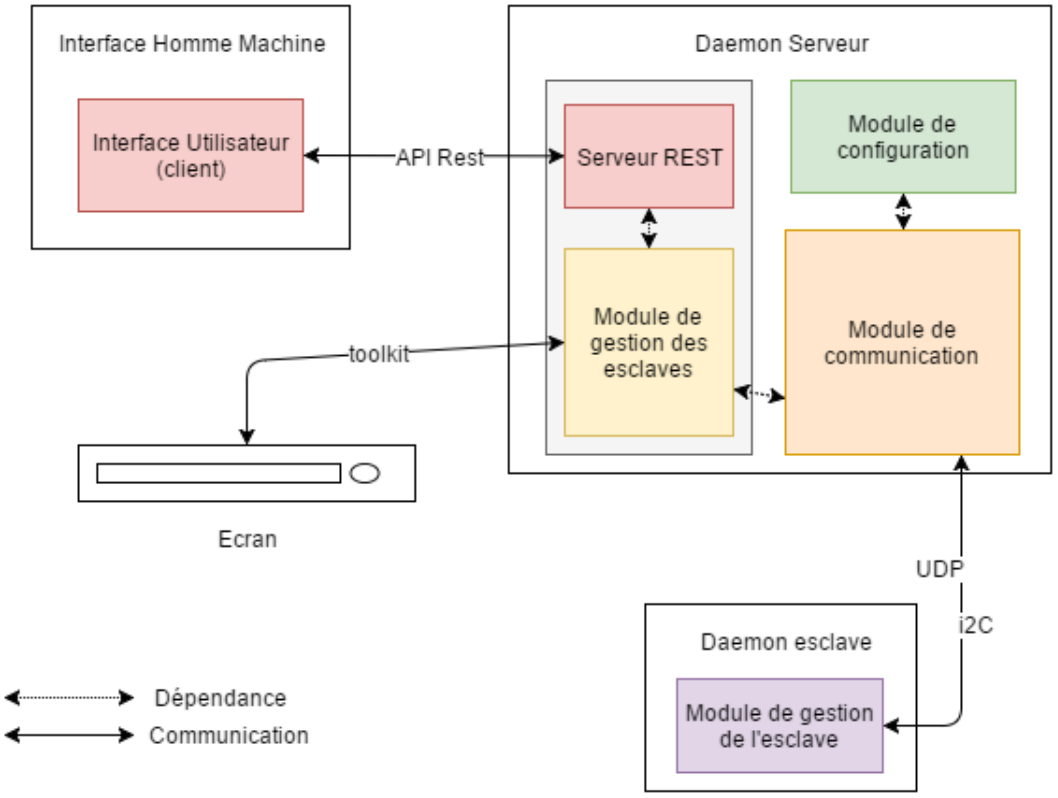

FIGURE 2 – Architecture globale de la solution logicielle

4.2 Installation de l'image

4.2.1 Génération des images utilisées

Une image sera générée à l'aide d'une image basique d'un système pour Raspberry Pi (type raspbian, pidora ...), que nous modifierons à l'aide d'un script. Ce dernier réalisera de manière automatique tout ce qui a rapport avec des "mises à jour", ajout du Daemon (et son lancement au démarrage), ainsi que d'autres ajustements. Le master utilisera une image basée sur raspbian-lite, qui contiendra un serveur DHCP, un interpréteur Python, les différentes bibliothèques et dépendances utilisées (en particulier Flask).

4.3 Configuration du cluster

Un slave est repéré à l'aide d'un quintuplet (adresse MAC, adresse , numéro de bus, numéro de board, position). Par exemple, Pour le cinquième slave du troisième board, qui est relié au second bus, on pourrait avoir le quintuplet suivant : (5E :FF :56 :A2 :AF :15, 1011101, 2, 3, 5).

4.3.1 Procédure de configuration

La procédure de configuration est la suivante :
— on éteint tous les *slaves*.
— on allume un *slave* et on déclenche un timer.
— on scrute les logs DHCP du master jusqu'à ce que l'on détecte l'adresse MAC ou jusqu'à ce que le timer atteint une valeur *timeout*.
— Si le master détecte bien cette requête DHCP, il calcule et envoie le quintuplet de configuration au *slave*.
— Le *slave* configuré télécharge son image pendant que l'on passe au *slave* suivant.

Il n'est pas possible de détecter la présence ou non d'un Raspberry sur un bus i2c donné, d'où la présence d'un timeout. La durée de ce timeout sera à définir par des tests de démarrage.

4.4 Protocole de communication i2c

Voici le fonctionnement du protocole i2c :

— Configuration : la configuration des Raspberries se fait grâce au protocole UDP.

— Initialisation : afin de communiquer avec le slave, le master doit d'abord initialiser la communication : pour cela il envoie l'adresse i2c sur 7 bits du slave avec lequel il veut communiquer et un dernier bit indiquant s'il veut envoyer ou recevoir des données (0 pour écrire et 1 pour lire). Le slave acquitte 0 si l'adresse est la sienne. L'adresse i2c étant due à la position du slave sur la board, cela ne permet pas de confirmer que l'on parle au Raspberry auquel on souhaite parler. Comme nous pouvons avoir au maximum 96 Raspberries et que nous souhaitons savoir si la machine à laquelle nous parlons n'a pas été déplacée, il suffit de leur attribuer à chacune une clé sur 8 bits.

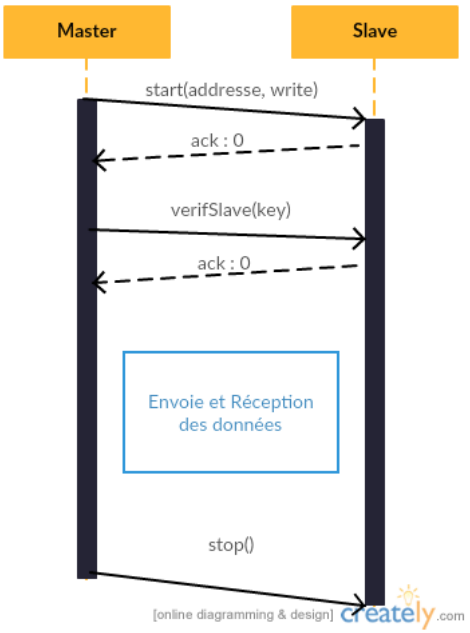

FIGURE 3 – Initialisation de la connexion

— Acquittement : Le mécanisme d'ACK et de NACK permet de savoir si un octet a bien été reçu ou non. Il est composé d'un bit. Nous avons donc décidé d'utiliser ce mécanisme afin de vérifier si notre clé a bien été reconnue ou non. Dans le cas ou elle n'est pas reconnue, cela veut dire que soit le slave n'a pas reçu la clé en entier, soit que celle-ci n'est pas la bonne. Dans le premier cas on renvoie la clé, et dans le second cas la procédure de configuration est relancée en UDP. Le diagramme de la Figure 4 suivante montre les deux cas.

— Envoi de données : Dans le cas où le master souhaite écrire, il peut envoyer des données par paquets de 8 bits. Le slave devra par la suite acquitter avec un bit tous les 8 bits (0 s'il a bien reçu les données, 1 s'il y a eu un problème). Voir Figure 5.

— Réception de données : dans la cadre de la récupération de données, le cas où le slave veut parler. Le slave pourra envoyer des messages par paquets de 8, ce à quoi le master répondra par un acquittement. Si l'acquittement est 0 alors le slave peut continuer à parler s'il le souhaite. S'il acquitte avec 1, alors la discussion est terminée, le slave n'a plus le droit de parler et le master ne recevra plus de données. Le slave enverra donc comme premier octet la taille des données qu'il souhaite envoyer afin que le master sache combien de temps et de données il doit lire. Ce ne sera pas le cas dans certaines situations comme la récupération d'erreurs car le master sait en avance combien d'octets il devra recevoir. Voir Figure 6.

— La fonction `restart` permet de modifier le statut lecture ou écriture d'une conversation. Ce mécanisme permet de ne pas avoir à refaire à chaque fois un `stop` puis un `start`, ce qui nous demanderait à chaque ouverture de faire la vérification du quintuplet. Cette fonction permet au master d'envoyer une commande au slave et de recevoir immédiatement une réponse de la part de ce slave.

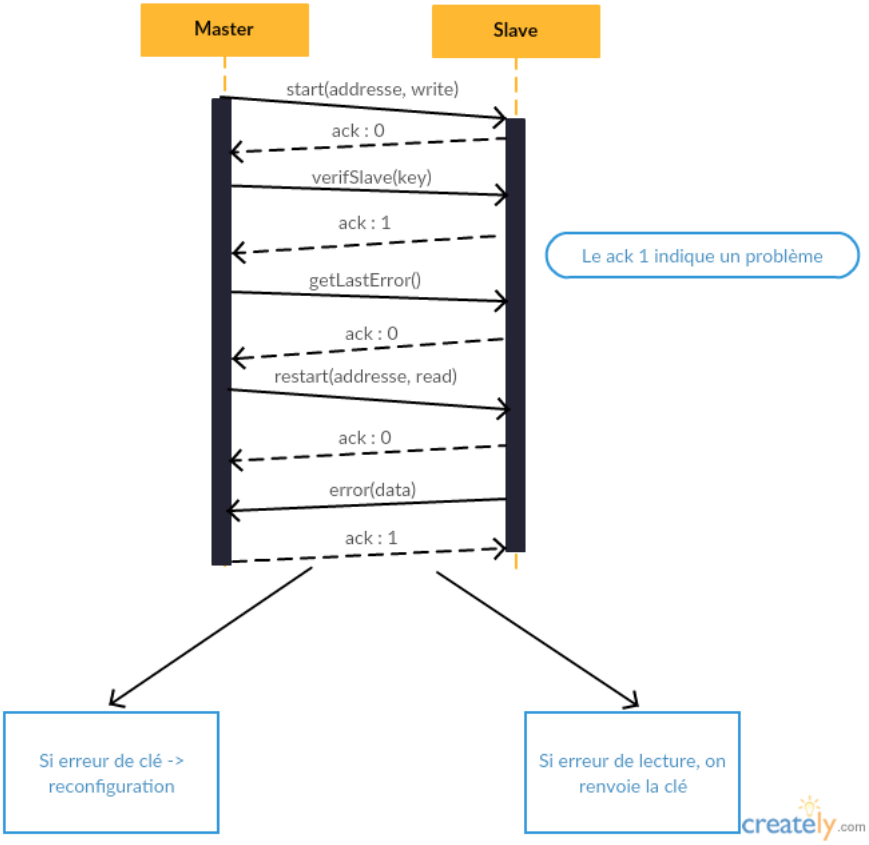

FIGURE 4 – Erreur à l'initialisation

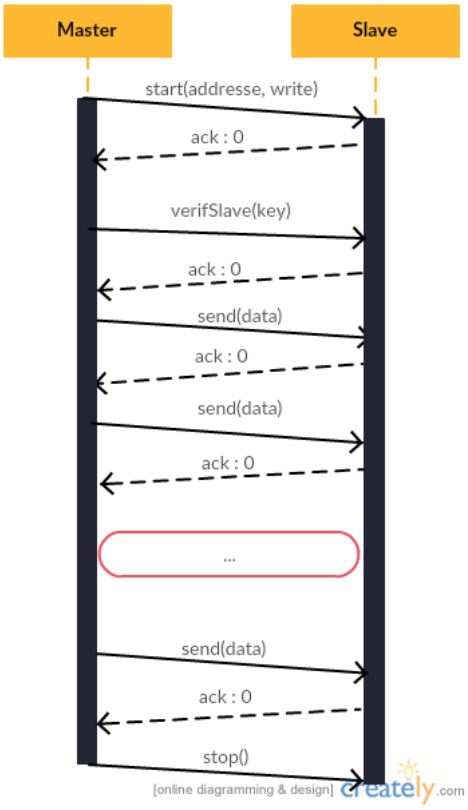

FIGURE 5 – Envoi de données du master vers le slave

— Exemple : la Figure 7 permet de voir quand le master envoie le fonction reboot à un slave.

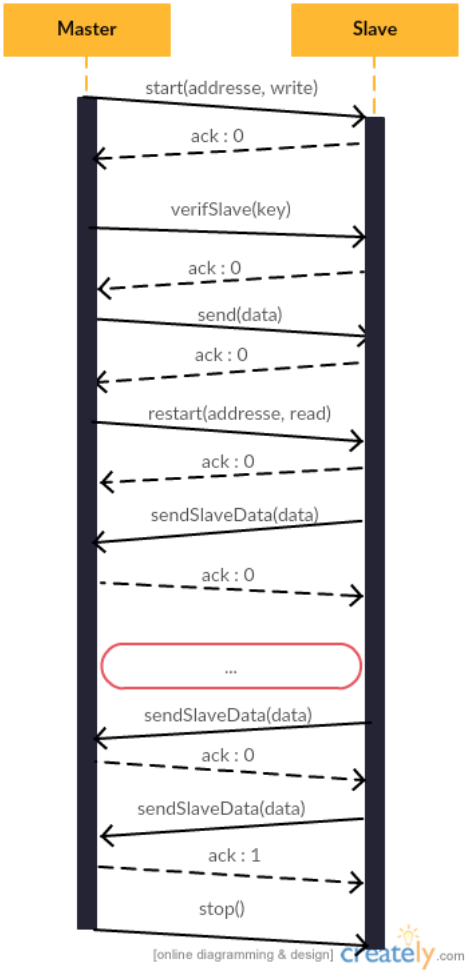

FIGURE 6 – Réception de données envoyées par le slave

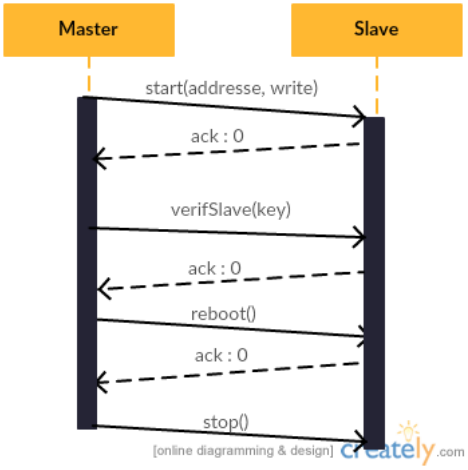

FIGURE 7 – Réception de données envoyées par le slave

4.5 Modélisation UML du cluster

À partir de la bibliothèque Python fournie qui sera utilisée par le master, un diagramme UML disponible en annexe a été réalisé. Cette bibliothèque contient notamment un certain nombre de valeurs hexadécimales dépendantes du schéma électronique. L'implémentation à réaliser inclut la modification de cette modélisation en la suivante (seuls les ajouts en vert sont détaillés pour plus de lisibilité) :

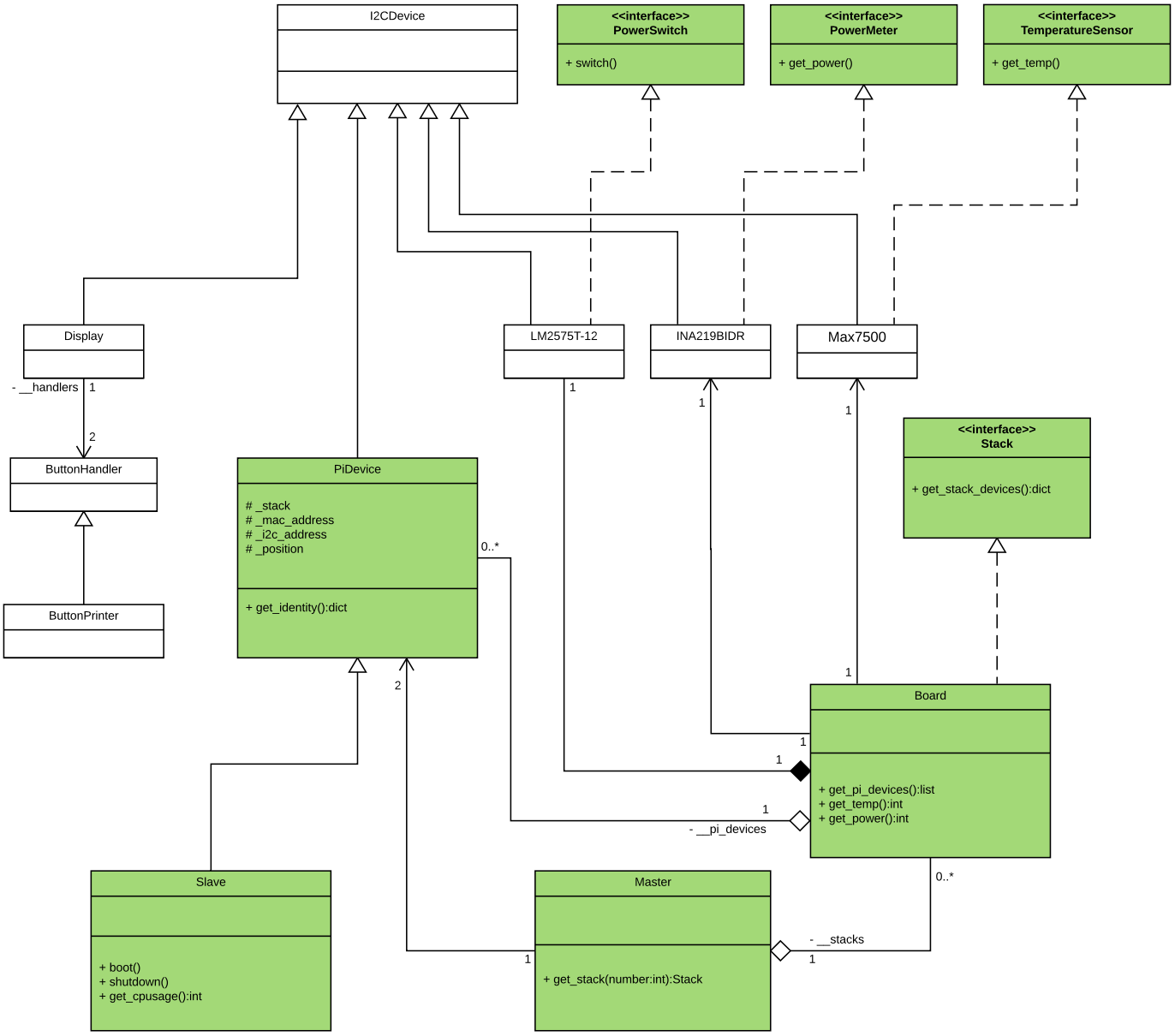

FIGURE 8 – Modélisation UML du cluster

La classe `Board` correspond à la représentation physique utilisée par le cluster. L'interface `Stack` permet de rendre évolutive l'implémentation du cluster en cas d'utilisation de stacks aux compositions électroniques hétérogènes. De même, les différentes interfaces `PowerSwitch`, `PowerMeter` et `TemperatureSensor` représentent les composants électroniques abstraits avec lesquels on peut interagir, en faisant abstraction des détails d'implémentation physique de ces derniers. Cela permet à nouveau une évolutivité du code. Ainsi, les classes héritant de ces interfaces représentent quand à elles les composants électroniques réellement utilisés dans le cluster, et implémenteront ainsi non seulement les méthodes inhérentes au type de composant(s) via l'interface, mais aussi des méthodes spécifiques à ce modèle précis.

`Board` est composé d'un`PowerSwitch` car il est obligatoire au fonctionnement du cluster. `Board` n'hérite pas de `PowerSwitch` car ce n'en est pas un, mais en contient un. Sans ce composant, le cluster ne peut pas fonctionner. Les deux autres composants `PowerMeter` et `TemperatureSensor` peuvent être désactivés sans que l'activité du cluster soit impactée.

4.6 Actions sur le cluster

Cette sous-section présente les différentes actions possibles sur le cluster. Comme expliqué dans les besoins non fonctionnels, chaque action correspond à une route de notre API Rest. Pour respecter la philosophie du protocole HTTP, la méthode GET est utilisée pour récupérer des données, la méthode POST pour émettre une commande ou envoyer des données, et la méthode OPTIONS pour lister ces actions et routes possibles. Ces actions seront d'abord implémentées en UDP puis si possible en i2c en utilisant le protocole de communication décrit ci-dessus. L'implémentation UDP servira de modèle pour prouver la validité de l'implémentation du protocole de communication i2c. Chaque action prend un paramètre `<id>`, correspondant à un numéro arbitraire de l'objet correspondant. S'il vaut `all`, alors l'action est effectuée sur tous les composants correspondants.

4.6.1 Actions sur les stacks

Les actions possibles sur une stack sont, avec les routes correspondantes :
— Obtenir sa consommation d'électricité en volts : `GET stack/power/<id>`
— Obtenir sa température en degrés Celsius : `GET stack/temp/<id>`
— Allumer ou éteindre toutes les Raspberries de la stack : `POST stack/shutdown/<id>`
— Obtenir les actions disponibles : `OPTIONS stack`

4.6.2 Actions sur les *slaves*

Les actions possibles sur un *slave* sont :
— connaitre le slave à qui l'on parle : `GET slave/index`
— obtenir son identité (quintuplet de configuration + adresse IP) : `GET slave/config/<id>`
— obtenir son taux d'utilisation CPU : `GET slave/cpu/<id>`
— allumer, éteindre ou redémarrer le*slave* : respectivement `POST slave/start, stop, restart/<id>`
— obtenir les actions disponibles : `OPTIONS slave`

Toutes les informations seront retournées sous format JSON, facilement traitable en Python (équivalent à un dictionnaire), et évidemment facilement traitable en Javascript pour l'IHM. Les routes pourront être implémentées en utilisant Flask-restful. Une session SSH sur un *slave* pourra donc être ouverte en récupérant son adresse IP.

4.7 Interface Homme Machine

L'interface se découpera en trois grandes parties :
— Une vue d'ensemble, qui décrira l'état actuel du réseau GrapeCluster avec une vue dynamique du réseau et la possibilité d'envoyer des commandes à l'intégralité des machines, en plus de pouvoir obtenir des informations de base (consommation de chaque stack, ...).
— Une vue axée sur les stacks qui décrira plus particulièrement leurs informations (température, consommation, nombre de slaves branchés), et permettront d'interagir de manière générale avec les slaves disponibles (envoyer une commande à un ou plusieurs slaves, avoir des informations basiques dessus).
— Une dernière vue spécifique au slave, décrivant un historique des commandes lancées dessus, des éventuels retours, la possibilité de leur envoyer des commandes ... C'est également ici que l'utilisateur aura la possibilité d'accéder au shell ssh sur le slave.

De plus nous utiliserons un système de notifications, pour permettre à l'utilisateur d'être au courant des retours de commandes, des erreurs ou des problèmes pouvant subvenir sur son réseau. Ainsi toute notification lui permettra d'accéder directement au noeud du réseau concerné, et lui offrira la possibilité de prendre la main dessus et de le régler.

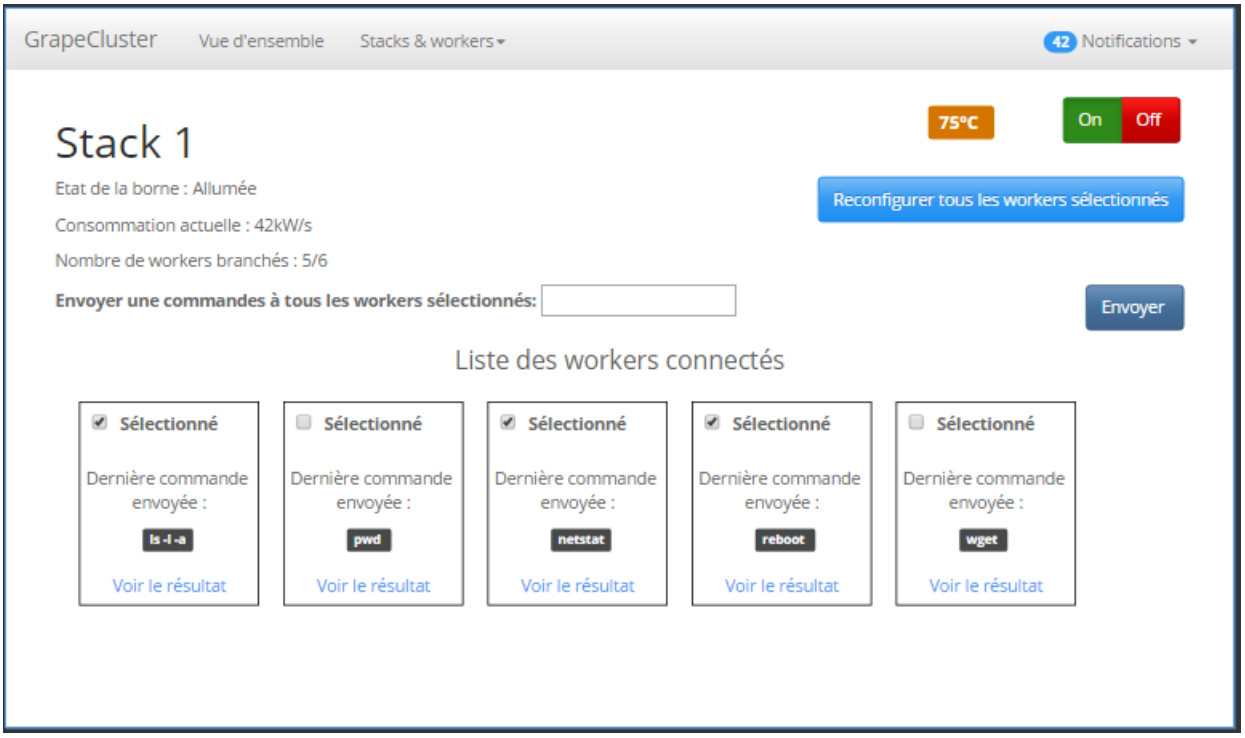

FIGURE 9 – Maquette simplifiée d'une stack

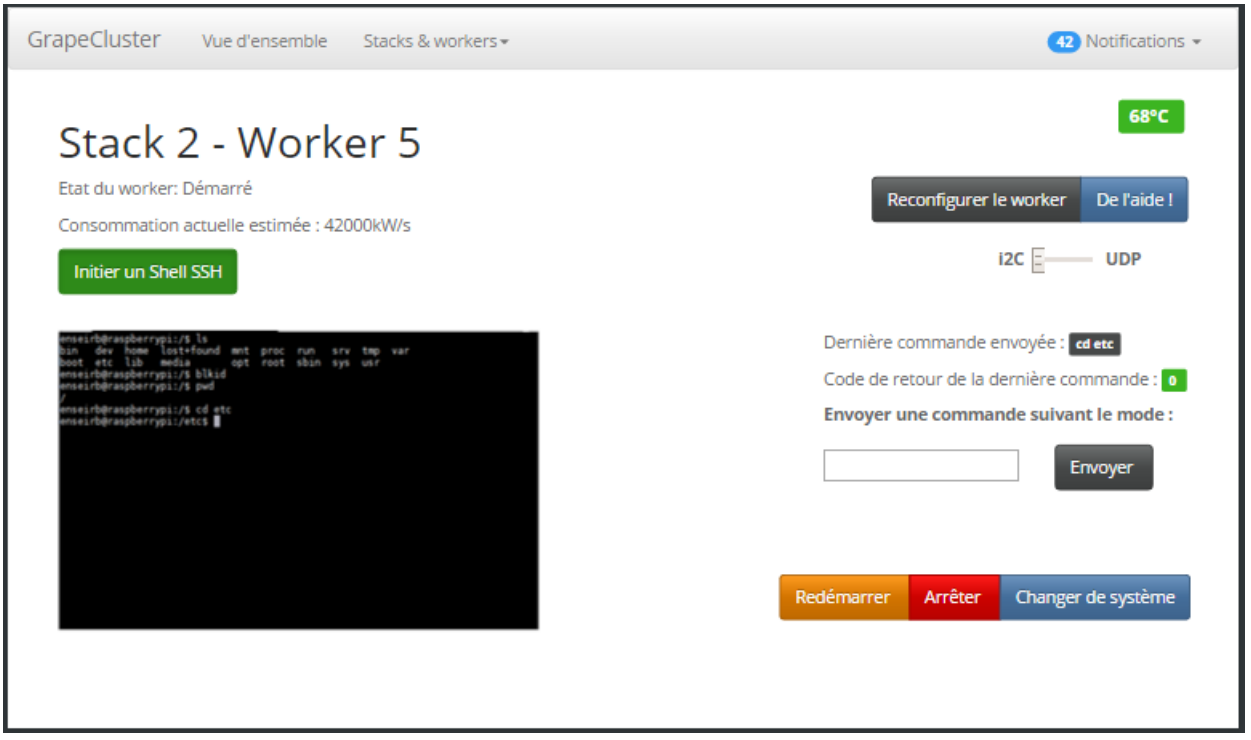

FIGURE 10 – Maquette simplifiée d'un slave

5 Plans de tests et validation

Tests réels grâce à 2 GrapesCluster-s pour l'installation d'un OS depuis le master sur un slave.

5.1 Test du protocole i2c

Afin de valider le protocole nous devrons réaliser les tests suivants :

Tout d'abord nous devrons tester la connexion entre les slaves et le master. Pour cela, nous réaliserons les tests de :
- Demande de connexion par le master
- Détection de la demande par un slave
- Envoi d'une clé erronée
- Retour du message d'erreur : clé erronée si tout c'est bien passé, message non reçu sinon

Puis nous testerons :
- Demande de connexion par le master
- Détection de la demande par un slave
- Envoi de la bonne clé
- Retour d'un ACK ou d'un message d'erreur : message non reçu.

Nous devrons ensuite, pour chaque commande implémentée dans l'idée d'être utilisée comme commande à envoyer par i2c tester :
- Envoi de la commande
- Si le slave reçoit bien la commande
  - Calcul de la réponse par le slave
  - Comparaison de ce que reçoit le master et de ce qu'a calculé le slave
  - Fermeture de la connexion
  - Vérification de la fermeture
- Si le slave ne reçoit pas la commande
  - Détection du protocole de gestion d'erreur

Enfin nous devrons tester l'envoi de données du master vers le slave.
- Envoi de données par le master
- Résutat de la différence entre données envoyées et données reçues égal à 0.

Les tests pourront être réalisés avec UDP afin de vérifier les différences entre les données sur le master et celles sur le slave.

5.2 Tests de fonctionnement de l'image modifiée

Pour construire et tester l'image modifiée avant de la flasher sur une carte SD sur laquelle démarrera un Raspberry Pi, le logiciel de virtualisation Qemu sera utilisé. Ainsi d'abord en modifiant l'image simplement en la montant puis en la virtualisant totalement nous pourrons tester du ou des résultats de modifications, d'imports et d'installations de packages. Puis en l'installant sur une carte SD nous pourrons finalement tester de la validité totale des installations.

6 Planning prévisionnel

Le diagramme de Gantt suivant présente le planning prévu. Chaque tâche présente implicitement des tests à réaliser, notamment lors des phases d'intégration entre plusieurs tâches principales.

Note : Le diagramme de Gantt a été découpé en deux parties pour plus de lisibilité. Le fichier original est joint à ce document.

Nom	Date de début	Date de fin
Installation d'un système sur une Raspberry	02/01/17	10/03/17
Ecriture du script d'installation d'une image	02/01/17	13/01/17
Implémentation du protocole de communication i2c	02/01/17	27/01/17
Prototypage du protocole i2c	02/01/17	11/01/17
Implémentation du protocole	12/01/17	27/01/17
Daemon du slave	25/01/17	23/02/17
Implémentation de la communication en UDP	25/01/17	03/02/17
Implémentation du Daemon et de ses interactions i2C/UDP	30/01/17	23/02/17
Installation et tests d'utilisation de l'image minimale sur le slave	16/01/17	24/01/17
Installation d'une image sur le slave (prête à être lancée)	24/02/17	10/03/17
Daemon du master	02/01/17	17/03/17
Serveur REST	02/01/17	10/02/17
Création des routes GET	02/01/17	27/01/17
Création des routes POST et OPTIONS	02/01/17	10/02/17
Processus de communication avec les slaves	02/01/17	17/03/17
Implémentation de la communication avec les slaves	02/01/17	24/02/17
Implémentation de l'i2c avec Python	02/01/17	26/01/17
Mise en place de la communication UDP	30/01/17	24/02/17
Mise en place des interactions avec les slaves	27/02/17	17/03/17
Interactions i2C	27/02/17	17/03/17
Interactions UDP	27/02/17	17/03/17
Interface Homme Machine	02/01/17	14/04/17
Prototypage de l'interface	02/01/17	27/01/17
Création d'une interface basique (GET)	30/01/17	24/02/17
Amélioration de l'interface avec les routes (POST et OPTIONS)	27/02/17	24/03/17
Intégration d'un Shell SSH	27/03/17	14/04/17

FIGURE 11 – Partie prévisionnelle du diagramme de Gantt

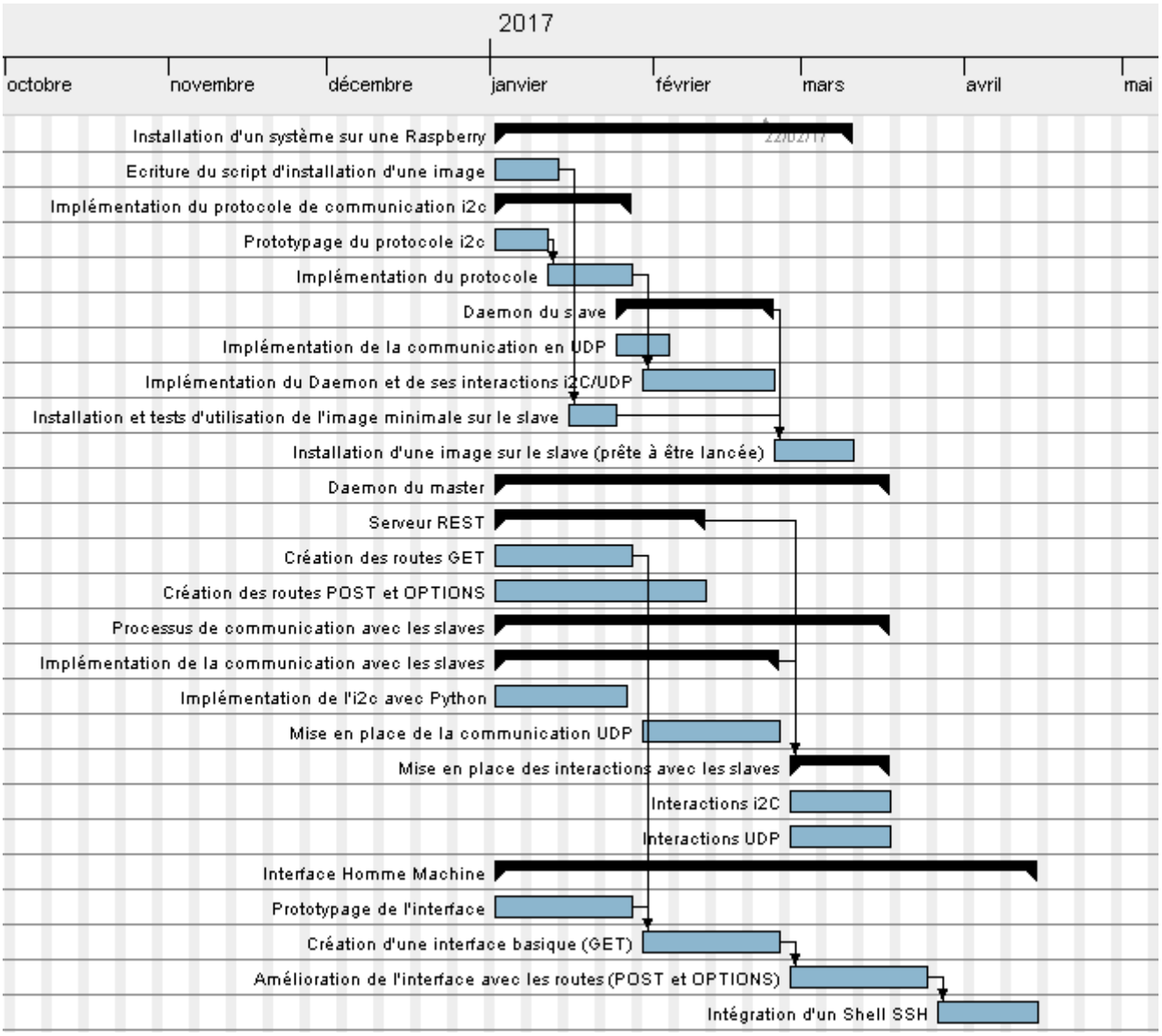

FIGURE 12 – Partie temporelle du diagramme de Gantt

7 Annexes

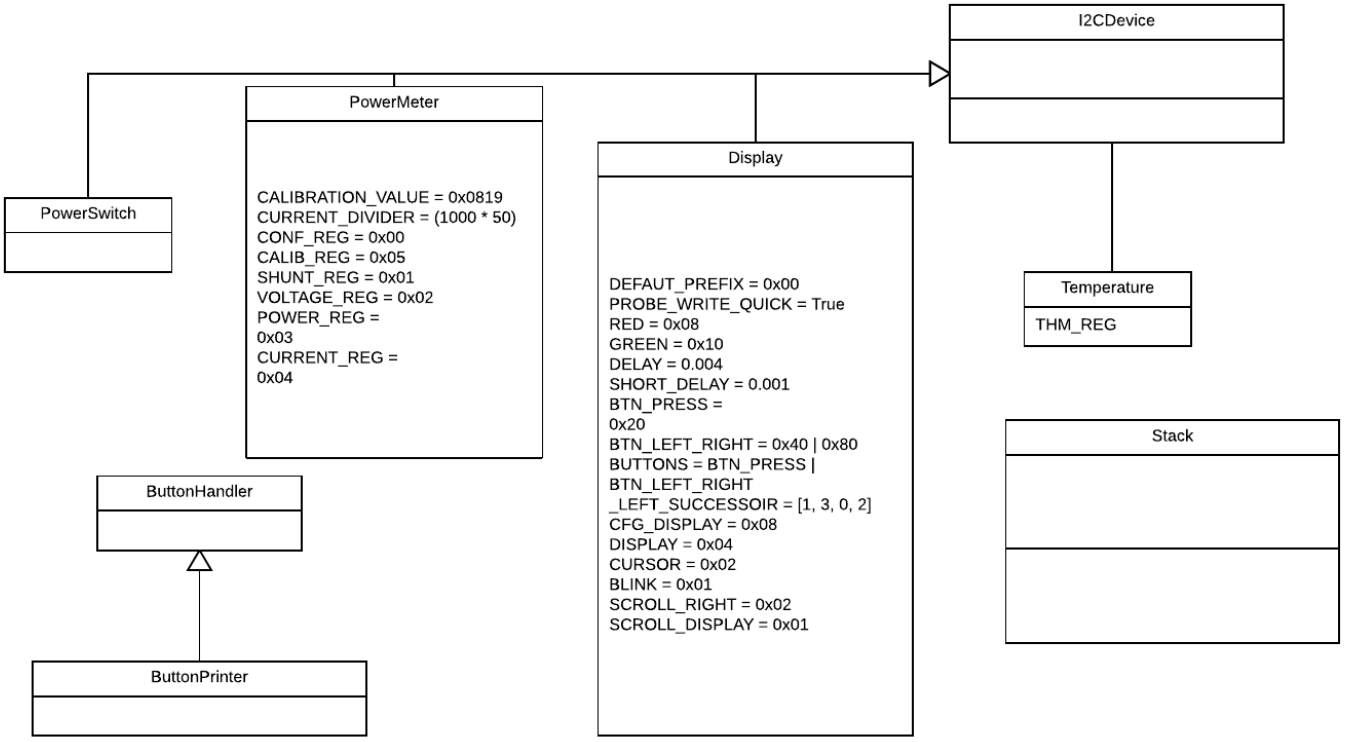

Figure 13 – Modélisation UML de la librairie fournie