PROJET IMA 4

SUIVI DE LA QUALITE DE L'AIR

DELBROUCQ Hugo
HAVARD Nicolas

IMA4

2018

SOMMAIRE

INTRODUCTION

De nos jours, l'environnement est un sujet important mais qui revient de manière récurrente : que faire pour améliorer l'environnement ? Comment savoir si l'air que nous respirons est bon ou mauvais ? Quels sont les facteurs influençant l'apparition d'un air nuisible et quelles en sont les conséquences sur la santé ?

Nous avons profité des projets proposés en quatrième année de notre cursus pour nous joindre au centre de recherche qu'est l'INRIA et qui travaille sur différentes problématiques du monde d'aujourd'hui et principalement tout ce qui a affaire nouvelles technologies telle que l'informatique.

En effet, l'INRIA proposait à l'un des binômes IMA de les rejoindre sur leur projet de suivi de la qualité d'air, projet consistant à récupérer des valeurs au travers de capteurs, que ce soit directement lié à la qualité de l'air comme le taux de monoxyde de carbone, de dioxyde de carbone ou de microparticules, mais aussi pour étudier les facteurs externes causant ces mauvaises conditions. Ces facteurs externes mesurés sont donc des facteurs environnementaux tels que la luminosité, la pression, l'humidité de l'air ou la température, mais aussi la présence de bruit ou de mouvement qui pourrait amener à une perturbation de la qualité de l'air due à l'être humain par exemple.

Intéressés par la cause environnementale et ses conséquences, mais aussi par l'aspect développement électronique mêlée à des milieux connus tels que les cartes Arduino et les micro ordinateurs Raspberry Pi, nous avons décidé de nous joindre à ce projet.

I. PRÉSENTATION DU PROJET

A. Maquette réalisée au cours de précédents projets

Tout d'abord, il est important de savoir que le projet n'est pas un projet parti de rien mais qu'il existe depuis un certain temps et que de nombreuses personnes ont contribué à sa mise en place. A notre arrivée, il y avait donc une maquette prototype déjà réalisée, disponible à l'INRIA. Cette maquette était principalement composée par une Raspberry Pi 2, d'une Arduino Mega ainsi que d'une panoplie de capteurs tels que les capteurs de mouvement, de bruit, de température, de pression, d'humidité ou de luminosité, mais aussi de capteurs Alphasense en grand nombre pour mesurer le taux de monoxyde de carbone, de dioxyde de carbone, ou encore le taux de microparticules dans l'air.

La maquette n'ayant pas pour but d'être portative, elle est reliée à une prise internet par câble et est alimentée grâce à une prise secteur. Cette maquette dispose aussi de modules permettant des connexions via le WiFi ou en réseau ZigBee.

B. Architecture logicielle

La maquette réalisée était de plus fonctionnel et captait déjà des valeurs en provenance des capteurs situés sur l'Arduino, et les renvoyant sur la Raspberry Pi afin de stocker les valeurs captées sur une base de données SQLite. L'architecture logicielle n'ayant pas été réalisée ou non communiquée par les groupes nous précédant, nous avons donc dû étudier sa structure ainsi que son fonctionnement à travers les programmes déjà existants. De ce que nous avons pu observer, il existe deux répertoires principaux : *datapoller* et *datasender* dont les missions principales sont de gérer la préparation et l'envoie des données (leurs formes etc).

Chacun d'eux est décomposée de la manière suivante : un répertoire qui comprend toutes les bibliothèques utilisées (liés aux capteurs ou autres) puis un autre répertoire comprenant tous les fichiers source qui permettent de mieux comprendre le fonctionnement du système et de le faire fonctionner. Il semblerait aussi qu'un début de communication via Lora ait été effectué mais sans information supplémentaire.

On en conclut alors que le fonctionnement de notre système est le suivant :

Pour ce qui est de la structure de fonctionnement avec de multiples noeuds, elle est similaire à celle ci mise à part que nous passons par un script composé d'un daemon qui demande aux bases internes aux rasbpi d'envoyer les données qui n'ont pas encore pu être envoyées vers influxDB.

Les différentes tables de la bases de données sont agencées de la manière suivante :

II. ENVOI DES DONNÉES SUR L'APPLICATION EN LIGNE

Nous avons commencé la conception d'un script Bash permettant d'envoyer les données sur le serveur afin de les regrouper sur influxDB et de pouvoir les traiter par la suite. Il sera placé dans le répertoire /etc/cron.hourly dans le cas où nous pourrions envoyer les données toutes les heures : les données seraient ainsi cumulées durant une heure en local dans une base de donnée SQLite, afin d'être envoyées toutes les heures sur le serveur. Ce script se décompose en plusieurs parties :

- les variables utilisées dans le script correspondant aux fichiers que le programme va lire ou écrire
- les variables qui vont être appelées par le script comme les noms des bases de données, les adresses URL sur lesquelles nous envoyons les données où les variables systèmes comme le nom de la machine
- les variables récupérant les données à envoyer et effectuant les requêtes sur l'adresse du serveur

Les différentes variables sont :

- db : la base de donnée sur laquelle se trouve les données nous intéressant
- db_request : fichier texte qui contiendra le résultat de la requête SQL sur la base de données $db pour récupérer les données à envoyer
- URL : l'adresse URL du serveur sur lequel nous souhaitons envoyer les données, et sur lequel se trouve l'application influxDB
- send_port : par défaut 8086. C'est le port d'écoute d'influxDB et sur lequel il nous faut envoyer les données pour que l'application les ajoute aux bases désirées
- app_port : par défaut 8083. C'est le port sur lequel il faut se rendre pour visualiser l'application influxDB et donc les données stockées sur la base en ligne
- database : la base de données stockant les données en ligne et sur laquelle nous envoyons les données mesurées par les capteurs du noeud
- nodename : le nom du noeud, soit l'identifiant permettant de distinguer l'émetteur des données sur la base

Concernant la réalisation du programme, nous nous y sommes repris à trois fois avant d'arriver à un résultat intéressant. Les idées qui ont été envisagées étant de réaliser une copie de la base de données à partir d'un dump avant de l'exploiter grâce à la

commande *cut* via un script SQL puis sans script, avant de récupérer directement les données et de les traiter sans passer par un dump pour alléger le script et la partie récupération des valeurs à envoyer à l'application.

Ces changements de structures du script auront eu pour incidence du temps de perdu à refaire des parties qui fonctionnaient pourtant pour les optimiser et donc de changer plus ou moins les parties suivantes pour qu'elles correspondent à cette nouvelle manière de faire.

La phase finale du script est découpée de la manière suivante :

A. Les variables du script

Comme dit plus haut, nous commençons par instaurer des variables au début du script afin de rendre ce dernier plus dynamique, et donc facilement paramétrable lors d'une nouvelle utilisation ou d'un changement tels que l'adresse de l'hébergement de l'application influxDB, les ports ou encore les bases de données de l'application.

De cette manière, il nous serait très facile de reprendre ce script pour envoyer les données d'une nouvelle base de données SQLite sur une base en ligne influxDB pour une utilisation plus médicale ou pour renvoyer les données d'un projet.

Il nous a donc été nécessaire de mettre en avant les variables *db* pour la base de données SQLite nous intéressant ainsi que les variables *URL*, *send_port*, *app_port* et *influxDB_database*. Nous avons de plus réalisé un script permettant de choisir les variables par défaut proposés par le script afin de changer facilement d'utilisation.

B. L’exploitation de la base de données SQLite

Vient ensuite la partie où nous traitons la base de données à étudier. Nous effectuons directement notre requête SQL dans le script, à savoir récupérer les identifiants des capteurs *sensor_identifier*, la valeur mesurée *data* et l'heure d'émission de la valeur, le timestamp, *poll_time* de la table *data*. Nous effectuons une jointure entre les tables *data* et *polls* sur le timestamp présent dans les deux tables.

Cette jointure nous permettra donc de filtrer les données déjà envoyées sur l'application en ligne grâce au flag *sent* de la table *polls*. Nous ordonnons ensuite les valeurs par horodatage croissant pour respecter l'ordre chronologique.

Le résultat de cette requête sera envoyé vers *$db_request*, variable pointant vers un document texte grâce auquel nous allons récupérer les valeurs à envoyer.

C. Les variables du script

Enfin, la majeure partie de ce programme est la récupération des données et leur envoi sur l'application en ligne influxDB, hébergée sur le serveur de l'INRIA. Nous commençons par réaliser une boucle qui va permettre la lecture du fichier contenant le résultat de la requête SQL et donc toutes les lignes de valeurs à envoyer.

Tout d'abord, pour chaque ligne, nous allons récupérer les noms du capteur ayant pris la valeur, la valeur en elle-même ainsi que le timestamp qui est en fait un horodatage. Le facteur 10^9 s'explique ici par le fait que l'application influxDB gère des timestamp en nanosecondes, alors que ceux-ci sont enregistrés en secondes seulement sur la base de données.

Pour envoyer les données sur l'application, nous disposons de la commande *cURL* va nous permettre de faire une requête HTTP en suivant la méthode POST (d'où l'argument -XPOST) et qui fonctionne de la manière suivante :

curl -XPOST *"adresse_influxDB:port_envoi****/write?db=****base_données_influxDB"* ***--data-binary*** ***"****table_influxDB,****hostname=****ID_noeud* ***value=****valeur_envoyée heure_de_la_valeur"*

Nous construisons donc d'abord l'adresse nous permettant d'envoyer les données sur l'application pour l'intégrer directement à la commande *cURL* puisque cette commande n'avait pas fonctionné lorsque nous y avions intégré les variables directement. Nous procédons de la même manière quant à la chaîne de caractères contenant les valeurs à envoyer. Afin de ne pas parasiter le déroulement du script par du visuel peu utile dans notre cas, nous appliquons l'argument *--silent*. De plus, l'argument --write-out complété à '*%{http_code}*' nous permet de récupérer le statut HTTP.

En effet, lors d'une requête HTTP, différents codes peuvent être renvoyés suite à l'envoi de données sur un serveur. Un code de 2XX (comprendre entre 200 et 299) signifiera un succès de la requête, 3XX une redirection de la requête, 4XX une erreur du client web et 5XX une erreur du serveur (les codes 1XX étant des informations). Nous souhaitons donc être sûrs que notre requête a bien fonctionné, autrement dit que influxDB a bien reçu les valeurs que nous avons envoyé. Pour cela, il nous faut recevoir un code 2XX, souvent 204 pour influxDB. Si ce code 204 n'est pas reçu, il sera nécessaire de temporiser et de recommencer l'envoi de la donnée jusqu'à ce que le serveur la reçoive.

Pour finir, nous avons instauré une progression dynamique lors du déroulement du script. En effet, lors du test de notre programme, ce dernier avait mis un peu moins de 3 minutes et 30 secondes pour envoyer 1000 valeurs sur le serveur. Ainsi, dans le cas où un opérateur souhaiterait envoyer quelques centaines voire milliers de valeurs, il lui faudrait attendre un temps considérable et il serait alors intéressant de montrer à l'utilisateur la progression du script, et que celui-ci avance correctement.
Nous avons donc installé le paquet *bc* qui permet de faire des calculs en Bash et gère ainsi les nombres flottants. Durant chaque envoi de valeurs, nous actualisons donc la dernière ligne du script (en effaçant celle d'avant afin d'obtenir un rendu plus propre) pour qu'elle affiche le nombre de données envoyées depuis le début du script, le nombre de valeurs à envoyer au total, et le pourcentage de progression que cela représente par rapport à ces nombres.

III. SOLUTION DE DÉPLOIEMENT DU RÉSEAU DE NOEUDS DE CAPTEURS

La solution pour faciliter le déploiement des différents noeuds a été proposé par nos encadrant. En effet, docker va ici nous servir à récupérer et installer des noeuds supplémentaires plus rapidement tout en se servant des variables d'environnement pour paramétrer chaque noeud de manière personnalisée.

Avant conteneurisation du projet, il nous faut d'abord tout placer dans un même répertoire et paramétrer le Dockerfile. Le Dockerfile va nous servir à créer une image personnalisée propre à notre projet qui va nous permettre d'exécuter notre conteneur dans de bonnes conditions. Notamment télécharger sqlite3, bc, python-dev, définir les variables d'environnement et préciser quels ports il nous faut laisser libre.

On construit maintenant notre image à l'aide de la commande :

"*docker build -t img_env_proj .*"

Une fois l'image créée, on l'importe sur le hub de docker afin de pouvoir aller la chercher à partir de n'importe quelle raspberry.

De plus, à l'aide de la commande "*docker container run -d --name node1 -e NAME=Node1"*, nous pouvons paramétrer nos variables d'environnement et ainsi personnaliser le nom de notre noeud.

Avec plus de temps lors du projet, l'un des objectifs était de moduler le positionnement des capteurs sur la maquette. Hors ces capteurs étant analogiques, il est impossible de déterminer quel capteur est à tel emplacement. Cela aurait pu être fait à l'aide d'un fichier de configuration nous proposant différents choix de configuration en fonction d'une variable d'environnement que nous aurions pu paramétrer de la même manière que le nom du noeud qui est affiché sur influxDB. De plus, on aurait aussi pu tout automatiser en ajoutant des services et autres ou mettre sous conteneur le serveur afin d'avoir notre relation Manager/Worker.

CONCLUSION

Ce projet nous a apporté beaucoup de recul sur ce qui sera à l'avenir notre métier en tant qu'ingénieur. S'inscrire à une démarche de projet n'est pas seulement traiter la commande d'un client en réalisant une maquett ou un programme grâce à nos compétences techniques, mais c'est avant tout une mission où les concernés doivent faire preuve de compétences managériales. Il est alors demandé de savoir s'organiser, établir un planning rapidement, gérer le temps et le retard en gérant le stress éventuel suite à une complication qui arrivera, différemment, mais dans tous les cas.

Sans avoir d'équipe à gérer, mais seulement un binôme, nous avons pu nous rendre compte les difficultés qu'imposent un travail aussi lourd sur la durée, et nous sommes donc heureux d'avoir grandi tout au long de cette période bien que nous ayons compris certaines choses trop tard.