

FONCTIONS LAMBDA POUR UNE PLATEFORME INTERNET DES OBJETS INDUSTRIELS DEDIEES A LA SECURITE INCENDIE

Rapport Final Du Travail De Bachelor



Tchente Simo Dany
HEIG-VD



Pr. Marcel Graf

Cahier des charges

Novaccess a développé une plateforme de gestion des objets connectés (Internet of Things, IoT). Cette plateforme est notamment utilisée dans une solution de surveillance de centrale incendie au moyen orient. Elle équipe des quartiers de villas, de la centrale incendie connectée par un réseau sans-fil maille, jusqu'à des passerelles réseaux installés dans des locaux de sécurité.

Actuellement le traitement et stockage des données se fait essentiellement dans le Cloud. La solution bénéficierait à avoir une partie fonctionnelle en local en plus du Cloud. Cela dans le cas où la connexion à longue distance avec le Cloud n'est pas disponible, et certains traitements doivent être faits en local (principe du Edge Computing).

Amazon a introduit une PaaS spécifiquement conçue pour développer des solutions IoT, nommé AWS IoT. Deux concepts essentiels de cette PaaS sont les fonctions Lambda et les Device Shadows. Amazon a aussi introduit une solution Edge Computing, nommée AWS Greengrass. C'est une plateforme logicielle qui tourne sur un Raspberry Pi et qui crée un environnement qui permet d'exécuter du code comme dans le Cloud sous la forme de fonctions Lambda. Aussi elle fournit des Device Shadows en local qui sont synchronisés avec le Cloud. Ainsi le Raspberry Pi crée un "mini Cloud" qui peut être déployé en local.

Le but de ce travail de Bachelor est de développer un prototype d'une solution de surveillance incendie qui repose sur AWS IoT, AWS Greengrass, les fonctions Lambda et les Device Shadows. Le prototype doit réaliser certaines fonctionnalités en fonction Lambda et les déployer dans le Cloud Amazon ainsi que sur un Raspberri Pi sur lequel AWS Greengrass a été installé.

Ce travail de Bachelor comprendra les étapes suivantes :

- Familiarisation avec les fonctions Lambda, les Device Shadows et AWS Greengrass
- Analyse fonctionnelle de la solution Novaccess pour la surveillance incendie
- Conception et développement d'un prototype d'une solution surveillance incendie selon du principe Edge Computing / Cloud Computing en utilisant les technologies AWS IoT et AWS Greengrass
 - Collection de mesures de divers capteurs (fumée, température, ...)
 - Stockage dans une base de données
 - Analyse des données pour détecter un incendie
 - Gestion des alarmes

Table des matières

Cahier des charges	1
Résumé	4
Remerciements.....	5
INTRODUCTION.....	6
Partie I : Expérimentation	7
I- ARCHITECTURE GENERALE.....	7
1- Connexion réseau.....	8
2- Transmission temporisée d'une alarme unique pour lever de doute local.....	8
3- Transmission directe de plusieurs alarmes simultanées.....	8
4- Partage du statut entre plusieurs passerelles en réseau local	8
II- FONCTIONS LAMBDA	8
1- Pré-étude	11
2- Expérimentation.....	12
III- CHOIX TECHNIQUES	18
1- Choix de l'équipement matériel	18
2- Choix du langage de programmation.....	19
3- Choix de la base de données.....	20
IV- CONCEPT EDGE COMPUTING	21
1- Composants AWS IOT.....	22
a- Passerelle machine to machine	22
b- Agent de messages.....	22
c- Service de sécurité et d'identité.....	22
d- Shadow d'appareil.....	22
e- Service Device Shadow	23
2- Stockage des données en local	23
Partie II : Conception et réalisation	30
I- Architecture détaillée de notre infrastructure.....	30
1- Phase de récupération et de stockage des données	32
2- Phase de traitement des données.....	32
a- Détection incendie	33
b- Détection d'un ou plusieurs capteurs défaillants	35
II- Réalisation de la partie capteurs.....	35

III-	Réalisation de la partie stockage des données via la fonction lambda et réPLICATION dans le cloud	36
IV-	Réalisation de la partie traitement des données.....	39
V-	Réalisation de la partie affichage.....	48
VI-	Bugs	52
VII-	Développements et améliorations futures	53
CONCLUSION		54
BIBLIOGRAPHIE.....		55
AUTHENTIFICATION.....		57

Résumé

Contexte et problématique :

Le monde d'aujourd'hui dépend fortement d'internet et il devient de plus en plus nécessaire d'avoir des objets interconnectés dans le cadre de l'industrie.

Ces objets interconnectés ont plusieurs domaines d'applications parmi lesquelles nous avons la gestion de l'éclairage public, la gestion des transports publics, les maisons intelligentes, la sécurité incendie, la santé et plein d'autres domaines que nous ne pouvons citer ici. Le domaine qui nous intéresse particulièrement est la sécurité incendie, dont une solution pour la surveillance des centrales incendie au moyen orient a été développée par une entreprise novatrice dans la télégestion des objets interconnectés. Cette solution consiste à l'utilisation d'une plateforme ayant pour but le traitement et le stockage des données dans le cloud.

C'est dans cette optique que l'entreprise Novaccess nous a mandaté afin de développer une plateforme de gestion des objets connectés en local et dans le cloud grâce à l'utilisation des fonctions lambda.

But du travail :

Le but de notre travail de Bachelor est de développer un prototype d'une solution de surveillance incendie qui repose sur AWS IoT, AWS GreenGrass, les fonctions Lambda et les Device shadows. Le prototype doit réaliser certaines fonctionnalités en fonction Lambda et les déployer dans le Cloud Amazon ainsi que sur un Raspberry Pi sur lequel AWS GreenGrass a été installé.

Résultats :

Nous avons réalisé une solution fonctionnelle effectuant un traitement et un stockage des données en local, mais ne permettant pas à un utilisateur de confirmer ou de quittancer une alarme détectée par nos capteurs en local. L'utilisateur a uniquement la possibilité de consulter la liste des capteurs ayant détectés cette alarme et la liste des capteurs défaillants si un ou plusieurs capteurs n'envoient plus de données.

Notre solution est avantageuse en comparaison à une solution déployée dans le cloud, dans la mesure où nous parons aux problèmes de perte de connexion avec le cloud de nos équipements, ce qui entraînerait des pertes de données et une baisse de performance dans le traitement et le stockage des données.

Une amélioration à notre solution serait de permettre à l'utilisateur de quittancer ou de confirmer des alarmes et enfin de développer une application web et mobile afin d'afficher nos données en temps réels pour une utilisation en entreprise.

Remerciements

Je remercie particulièrement le Professeur Marcel Graf pour m'avoir confié ce sujet et pour m'avoir suivi avec une grande attention. Il a partagé avec moi son expérience et m'a permis de mieux comprendre certains concepts qui m'ont permis de m'immerger davantage dans la résolution du problème qui m'était posé.

Je remercie aussi ma famille, mes amis et ma copine qui m'ont épaulé et soutenu mentalement durant mes longues journées de travail intensif.

INTRODUCTION

De nos jours les avancées technologiques dans le domaine de l'internet des objets tendent à faire connecter plusieurs objets entre eux afin d'optimiser les performances et le rendement des services.

C'est dans cette même optique qu'il nous a été proposé par l'entreprise Novaccess de réaliser une plateforme Internet des Objets Industriels dans le but de gérer la sécurité incendie via des fonctions lambda. Les fonctions lambda représentent une toute nouvelle approche dans la conception des applications. Elles permettent de déployer une seule fonction sur un serveur dans le cloud sans se soucier de la gestion et de la disponibilité du serveur ainsi que la facturation correspondant au temps d'exécution de la fonction lambda. Cette architecture porte le nom de « Function-as-a-Service » ou architecture serverless.

Étant donné que Novaccess dispose d'une plateforme utilisée comme solution pour la sécurité incendie dans le cloud, il nous a été demandé de développer plus précisément un prototype de cette solution en local et dans le cloud. Cette solution est implémentée en utilisant la plateforme d'Amazon dédiée aux solutions IoT à savoir AWS IOT. Cette plateforme AWS IOT permet le déploiement d'un mini cloud sur un RASPBERRY PI sur lequel est exécuté une ou plusieurs fonctions lambda et sur lequel est connecté un ou plusieurs appareils.

Pour réaliser notre travail nous l'avons divisé en deux parties :

- Une partie expérimentale dans laquelle nous explorons les différents services fournis par Amazon afin de sélectionner ceux qui sont utiles à la résolution de notre cahier des charges. Cette partie se subdivise en quatre chapitres à savoir une architecture générale de notre prototype, une étude des fonctions Lambda, ensuite les choix techniques concernant le langage de programmation, la base de données et le matériel utilisés et enfin la compréhension du concept d'« edge computing » concernant le déploiement en local de notre traitement et stockage des données.
- Une partie conceptuelle dans laquelle nous concevons notre architecture et utilisons les outils (services Amazon et autres) choisis pour la réalisation de notre solution. Cette partie quant à elle se subdivise en cinq chapitres à savoir la réalisation d'une architecture détaillée de notre infrastructure, la réalisation de la partie capteur afin de récupérer les valeurs fournies par les capteurs, ensuite la réalisation de la partie stockage dans le but de stocker nos données lues par les capteurs dans une base de données, la réalisation de la partie traitement de nos données en local et enfin la réalisation de la partie affichage des données pour la gestion des alarmes.

Partie I : Expérimentation

Il est question dans cette partie de nous familiariser avec la plateforme d'Amazon et les services offerts par celle-ci qui nous permettront d'atteindre nos objectifs et de concevoir la solution à notre problème. Nous explorons aussi dans cette partie d'autres services et plateforme afin de sélectionner ceux qui peuvent nous être utile.

I- ARCHITECTURE GENERALE

Avant de nous lancer dans la recherche et la compréhension des services d'Amazon, il est important d'avoir une idée générale de notre architecture. Le schéma ci-dessous présente cette architecture dans laquelle nous remarquons chaque composant à savoir les villas, les capteurs, la passerelle et le serveur Nora qui est remplacé dans notre cas par notre application déployée dans une instance pour un accès à distance. Nous décrivons dans cette partie les différents composants de cette architecture.

Chaque capteur installé dans une villa, est connecté à un réseau maillé. Tous ces capteurs sont reliés à une passerelle correspondant à notre RASPBERRY PI dont le rôle est de recevoir, traiter et sauvegarder les données des capteurs dans notre base de données (Couchdb). Ce traitement est réalisé par une fonction lambda déployée sur notre passerelle. Les données sauvegardées dans la base de données sont synchronisées avec celles contenues dans la base de données (Couchdb) installée sur le serveur Nora et sur le serveur de la défense civil de Dubaï.

Nous avons plusieurs passerelles installées chacune dans des locaux de surveillance et on distingue des locaux dans lesquels les gardes pourront être soit absents soit présents de manière permanente.

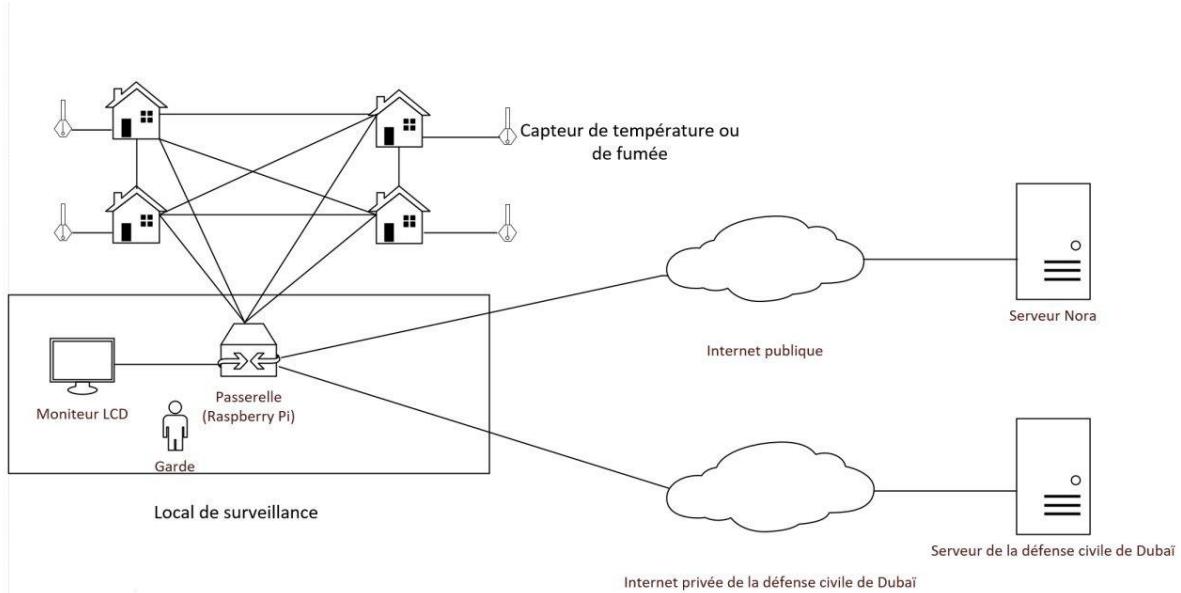


Figure 1: Architecture générale de notre projet

Afin de démontrer le bon fonctionnement de cette architecture, il nous a été demandé de réaliser des scénarios de démonstration à savoir :

1- Connexion réseau

Dans ce scénario il est question de connecter nos objets au réseau sans-fil maillé représenté sur la « figure 1 ». Pour ce faire, il faut associer chaque équipement au réseau et paramétriser les autorisations dans l'optique que nos équipements ne puissent pas interagir directement avec le serveur Nora mais plutôt avec la fonction lambda déployée sur la passerelle.

2- Transmission temporisée d'une alarme unique pour lever de doute local

Il est question dans ce deuxième scénario de simuler une détection d'incendie dans une des villas. En effet les données générées par nos capteurs (température et humidité), sont transmises à notre passerelle pour traitement par notre fonction lambda. Ces données sont ensuite affichées sur un écran dans le local de garde afin qu'un gardien puisse annuler ou confirmer la détection d'incendie.

Cependant si l'alarme n'est pas annulée dans les deux minutes suivant sa détection, elle est automatiquement transmise vers le serveur Nora et celui de la défense civile. Mais si l'alarme est annulée, alors le gardien peut à l'aide d'un bouton, désactiver l'effet sonore généré par l'alarme.

3- Transmission directe de plusieurs alarmes simultanées

Dans ce scénario, il est question de configurer notre fonction lambda dans le but de transmettre automatiquement les alarmes sur le serveur Nora dans le cas d'une détection simultanée.

4- Partage du statut entre plusieurs passerelles en réseau local

Il est question dans ce scénario de synchroniser les données recueillies par toutes les passerelles afin de fournir une meilleure redondance des données, permettant ainsi une intervention rapide sur le lieu de l'incendie.

II- FONCTIONS LAMBDA

Avant de nous lancer dans l'étude et l'expérimentation des fonctions lambda, il est question pour nous de définir ces fonctions, d'expliquer leur fonctionnement, leur utilisation et leur facturation.

En effet une fonction lambda est un service fourni par Amazon en tant que Function-as-a-Service dans le but d'exécuter un code ou un bout de code uniquement si un évènement est reçu. Afin de mieux comprendre ce service, il est important de comprendre les différents

modèles proposés par des services cloud à savoir Software-as-a-Service, Platform-as-a-Service et Infrastructure-as-a-Service.

Le premier modèle permet l'utilisation d'applications depuis un navigateur. Comme exemple, nous avons Office Online, Dropbox et Salesforce.

Le deuxième modèle permet aux utilisateurs de créer des applications dans le cloud en utilisant les outils, les ressources et le matériel fournis par le fournisseur de service cloud. Dans ces deux cas de figures précités, l'utilisateur n'a pas à se soucier de l'installation des outils nécessaires à l'exécution ou aux ressources matérielles et logicielles utilisées, tout est géré par le fournisseur de service.

Le troisième modèle permet aux utilisateurs de configurer les caractéristiques de la machine virtuelle, d'y installer les outils de développement, d'exécution et de déploiement de ses applications.

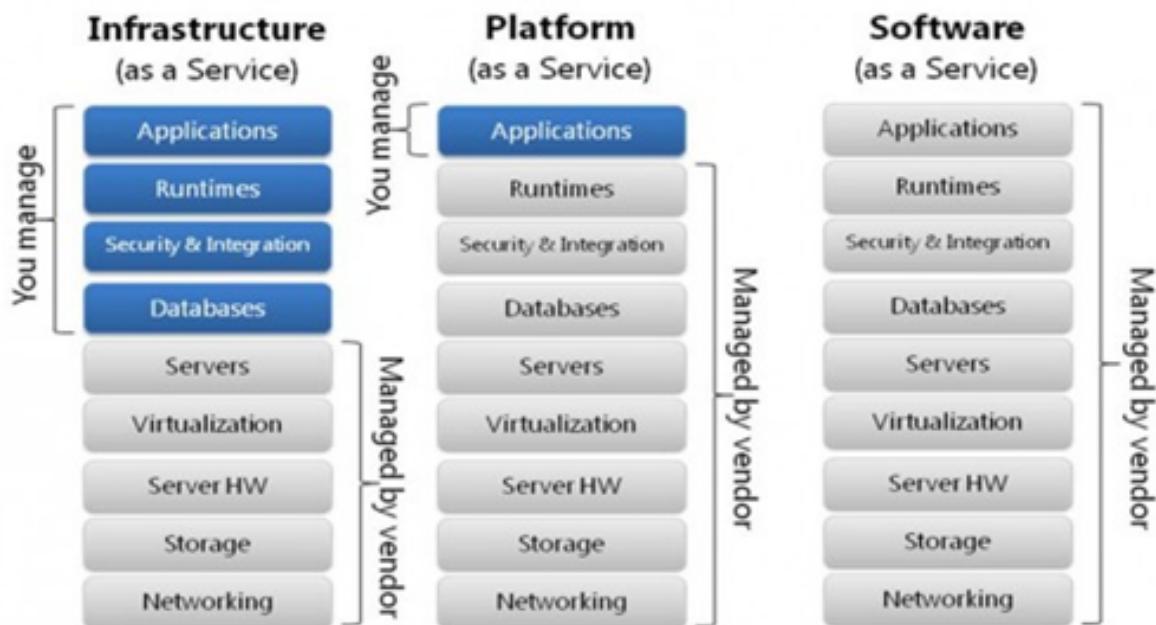


Figure 2 : Différence entre les modèles de service cloud en fonction des parties gérées par le fournisseur de service cloud et celles gérées par le client [1.3.6]

A ces différents modèles vient s'ajouter notre modèle « Function-as-a-Service (FaaS) » qui est presque similaire au modèle « Platform-as-a-Service » à la différence qu'au lieu de déployer toute une application, on ne déploie qu'une fonction de notre application réduisant ainsi le temps d'exécution de notre fonction et par le même biais, le coût financier généré à chaque appel de notre fonction, offrant aussi une meilleure modularité en fonction du nombre de requêtes reçues.

Un autre aspect important de ce modèle réside dans le fait que la durée de vie peut être configurée soit à « longue durée », soit à la « demande » en fonction de leur utilisation. Dans le cas d'une fonction à la demande, un contexte d'exécution est créé à chaque appel de la

fonction or dans le cas d'une fonction longue durée le contexte d'exécution est créé au premier démarrage de la fonction et reste le même durant toute la vie de la fonction. Le schéma ci-dessous nous montre les différents modèles de service fourni dans le cloud.

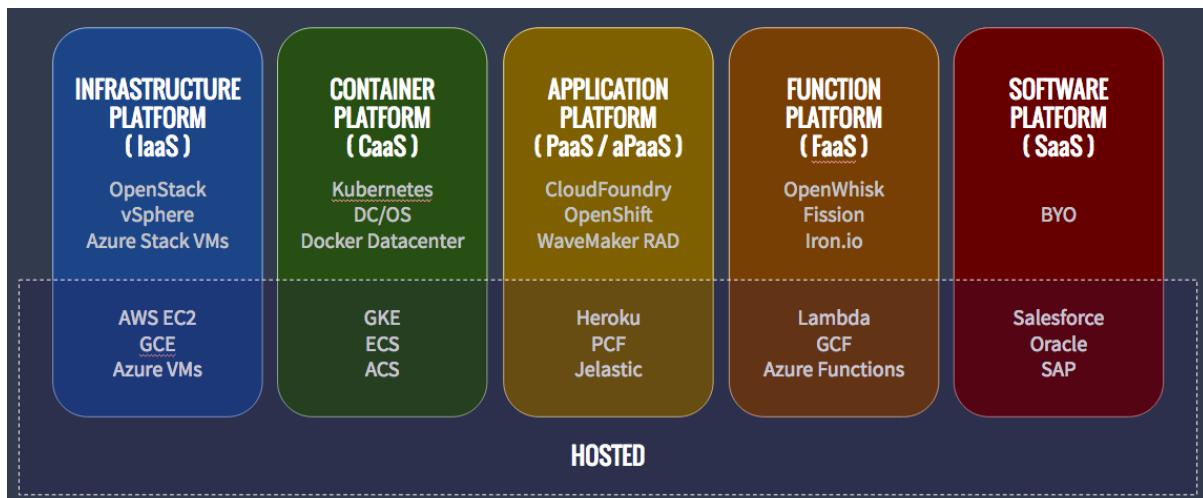


Figure 3 : Différents modèles de service cloud situant les Function-as-a-Service parmi les autres services cloud [1.3.5]

Quant aux évènements reçus par nos fonctions, ils proviennent d'autres services Amazon à savoir AWS DYNAMODB, Amazon S3, Amazon LEX, bouton AWS IOT, Amazon API Gateway et d'autres services que vous trouverez dans la documentation [1.3.7]. Ces fonctions une fois déployées ne nécessitent aucune administration de la part des utilisateurs. Comme exemple, nous pouvons citer les instances dans lesquelles le code est déployé, l'installation d'un système d'exploitation sur l'instance, la sécurisation de l'accès aux ressources, les caractéristiques matérielles de l'instance et des ressources allouées pour son exécution.

Ce service garantit une haute disponibilité et modularité dans le cas des requêtes reçues ou émises (chaque requête est traitée indépendamment d'une autre et ces ressources sont adaptées en fonctions du nombre de requête reçue) ainsi qu'une haute performance dans l'exécution du code.

S'agissant de la facturation, elle est calculée en fonction du temps d'exécution de la fonction et du nombre de requêtes reçues (arrondi à 100 ms par requête).

Cependant il existe d'autres fournisseurs de services qui proposent ce type de service (Function-as-a-Service). Ci-dessous, nous présentons un tableau comparatif de ce service en fonction de chaque fournisseur.

AWS Lambda	Langages supportés	Déploiement	Triggers	Temps exec max	Prix (1 Go RAM)
	NodeJS Python Java C#	Zip	Amazon Services API Gateway	5 minutes	Gratuit: 1M requêtes/mois 400k Go-sec/mois Payant: 0.20\$/1M requêtes 0.00001667\$/GB-s
	NodeJS	Zip Google Repo	Cloud Pub/Sub Cloud Storage Requêtes HTTP	9 minutes	Gratuit: 2M requêtes/mois 400k Go-sec/mois Payant: 0.40\$/1M requêtes 0.00001650\$/GB-s
	JavaScript Python C# PHP	Éditeur Web	MS Cloud Services Requêtes HTTP	5 minutes	Gratuit: 1M requêtes/mois 400k Go-sec/mois Payant: 0.20\$/1M requêtes 0.000016\$/GB-s

Figure 4: Tableau comparatif du service (Function-as-a-service) en fonction des différents fournisseurs de service cloud [1.3.8]

Pour se familiariser avec ce concept et comprendre son fonctionnement, afin de l'intégrer à notre problème, nous avons réalisé une pré-étude, ensuite une expérimentation de ces fonctions.

1- Pré-étude

Après avoir défini ce qu'est une fonction lambda, nous avons effectué une pré-étude qui consiste dans un premier temps à consulter la documentation fournie par Amazon afin d'apprendre davantage sur le service AWS LAMBDA. Comme présenté sur la capture ci-dessous, nous avons réalisé la première étape dans la section « Mise en route » d'AWS LAMBDA afin de comprendre les principes de base de la création des fonctions lambda et nous avons successivement :

- Configuré un compte AWS en s'inscrivant à AWS et en créant un utilisateur IAM (Identity and Access Management)
- Configuré l'interface ligne de commande AWS CLI afin de pouvoir utiliser le service AWS LAMBDA via un terminal et effectuer des actions comme créer une fonction lambda, lister nos fonctions lambda et etc ...
- Nous n'avons pas installé un SAM (Serverless Application Model) local car le test de nos fonctions lambda sur notre machine n'était pas nécessaire, étant donné que nos fonctions lambda sont déployées sur un RASPBERRY PI modèle 3B.
- Créé une fonction lambda « Hello from lambda » en suivant les étapes décrites dans la section « création d'une fonction lambda simple ». Dans cette section, nous nous sommes familiarisés avec les outils de test (analyse des logs) et de surveillance

(cloudwatch pour la journalisation) des fonctions lambda, le langage utilisé (python2.7), et les stratégies de sécurité d’AWS LAMBDA.

Après avoir compris les principes de base de la création d'une fonction lambda, à savoir sa configuration, sa création et son utilisation, nous avons réalisé la deuxième étape qui nous a permis de mieux comprendre le fonctionnement des fonctions lambda plus précisément comment elles sont exécutées dans le cloud, dans le but d'adapter leur fonctionnement à notre problème. Pour le réaliser, nous avons exploré plus en profondeur les sections suivantes de la documentation dont le détail vous est mentionné ci-dessous.

Nous nous sommes intéressés à la section « Fonctions Lambda », ce qui nous a permis par la suite de choisir un modèle de programmation Python (nous expliquerons le choix de ce langage dans le chapitre « Choix technique »), de créer et de configurer une fonction lambda via la console fournit par AWS LAMBDA pour la création et la configuration des fonctions lambda en ligne.

Dans la sous-section « Modèle de programmation » contenu dans la section « Fonction Lambda », il est expliqué la création, la journalisation et les erreurs éventuelles d'une fonction lambda codée en python.

Pour mieux comprendre l'exécution des fonctions lambda, il faut se référer à la section « Modèle d'exécution AWS LAMBDA ». Cette section nous a permis de comprendre la création et la suppression d'un contexte d'exécution, la gestion des ressources allouées au lancement et à l'exécution d'une fonction.

Après avoir compris le fonctionnement des fonctions lambda en général, nous avons réalisé la troisième étape intitulé « Appel de fonctions Lambda » afin de comprendre comment nos fonctions sont appelées, comment elles sont déclenchées et les sources d'événements.

Comme dernière étape, nous avons parcouru les sections « utilisation », « déploiement », « surveillance » et « administration » des fonctions lambda pour avoir une vue globale du déploiement, du suivi de nos fonctions lambda et de leur exécution afin d'y apporter des modifications et améliorations. Pour obtenir plus de détails veuillez consulter [1.3.1].

2- Expérimentation

Cette partie a pour but d'expérimenter et de manipuler les fonctions lambda afin de mettre en pratique les connaissances acquises lors de notre pré-étude. C'est dans cette optique que nous avons créé notre fonction lambda à partir des exemples fournis dans la documentation afin d'effectuer des tests et observer son exécution grâce aux outils fournis à savoir la journalisation.

Étant donné que nous aimions stocker les données reçues de divers capteurs, il est important de comprendre comment une fonction lambda peut interagir avec d'autres services d'Amazon (AWS DYNAMODB, AWS SNS) afin d'y stocker nos données. Notre fonction a donc pour but d'écrire des données reçues d'un ou plusieurs dispositifs (capteurs) dans notre table « FireData » contenu dans la base de données DYNAMODB qui est un service d'Amazon

proposant une base de données NoSQL. Pour avoir plus d'informations sur le lancement et le fonctionnement de AWS DYNAMODB, nous avons consulté la documentation [1.3.2]. Les captures ci-dessous correspondent à notre code écrit en python.

```

55 ▲ """  

56     @summary:      Cette méthode permet d'écrire des données dans la table  

57             FireData  

58  

59     @param points:   event: données d'événement dans lequel on retrouve les données  

60             à écrire dans la table  

61             context: informations d'exécution de notre fonction  

62  

63     @returns :      -----  

64 ▲ """  

65 ▼ def function_handler(event, context):  

66  

67     # récupération des données depuis le paramètre event  

68     # schema JSON:  

69     # { "time":DateTime, "value":number}  

70     logger.info(event)  

71     time = event["time"]  

72     value = event["value"]  

73     logger.info("time state: " + str(time))  

74     logger.info("value state: " + str(value))  

75  

76  

77  

78     # mis à jour de la table FireData dans dynamodb  

79     global tableName  

80     table = dynamodb.Table(tableName)  

81     table.put_item(  

82         Item={  

83             'time':time,  

84             'value':value  

85         }  

86     )  

87     return  

88

```

Figure 5 : Bout de code de notre fonction lambda permettant l'extraction et la sauvegarde des données dans la base de données DYNAMODB. Ces données correspondent à des valeurs aléatoires de températures relevées dans une villa à un moment (time) donné. Ce temps est sauvegardé suivant le format POSIX afin de le rendre générique en fonction du fuseau horaire d'utilisation.

La fonction lambda ci-dessus est composée d'une partie principale appelée « main » et d'une autre partie gestionnaire de fonction « function_handler ». La partie principale est exécutée une seule fois s'il s'agit d'une fonction ayant une durée de vie continue ou à chaque appel s'il s'agit d'une fonction à la demande. Dans notre cas de figure nous l'avons configurée à longue durée afin de créer la base de données une seule fois et de recevoir les données en continu de nos différents capteurs.

Quant à la partie gestionnaire de fonction, elle reste à l'écoute et lorsqu'un évènement est reçu, elle effectue le traitement associé. Dans la partie principale de notre fonction lambda, nous effectuons une connexion avec la base de données. Nous la sélectionnons si elle existe et nous

en créons une nouvelle si elle n'existe pas. Dans la partie gestionnaire de fonctions, nous récupérons les données reçues dans le paramètre « event » suite à une publication de ce payload dans la rubrique « fire/test/cloud » et ensuite nous les écrivons dans la base de données « fire_data ».

Pour réaliser ce traitement des données reçues par notre fonction et leur sauvegarde sur le cloud, nous avons utilisé comme fonction de base « carAggregator.py » qui est une fonction fournie dans les étapes de familiarisation avec les fonctions lambda contenues dans la documentation d'Amazon [1.3.3]. Nous avons ensuite modifié la structure JSON des données écrites dans DYNAMODB et supprimé des lignes inutiles. Nous avons aussi analysé le code original afin de mieux comprendre le fonctionnement et de pouvoir y apporter des modifications.

En effet l'importation du package boto3 n'est pas nécessaire dans la mesure où notre fonction sera déployée dans le cloud. Mais dans notre cas, comme nous allons le voir dans le « chapitre IV », notre fonction sera déployée sur un noyau greengrass tournant sur notre RASPBERRY PI.

Ce package boto3 est un kit SDK fournit par Amazon et nous permet d'utiliser d'autres services AWS tel que DYNAMODB dans notre cas précis. Nous avons de ce fait installé le package boto3 dans un dossier sur notre machine et y ajouté le code de notre fonction, dans le but de le déployer sur le cloud. La capture ci-dessous présente le contenu du dossier avec notre fonction portant le nom de « test.py ». Dans ce dossier nous avons importé le package boto3 comme présenté sur la figure 2 correspondant à notre code.

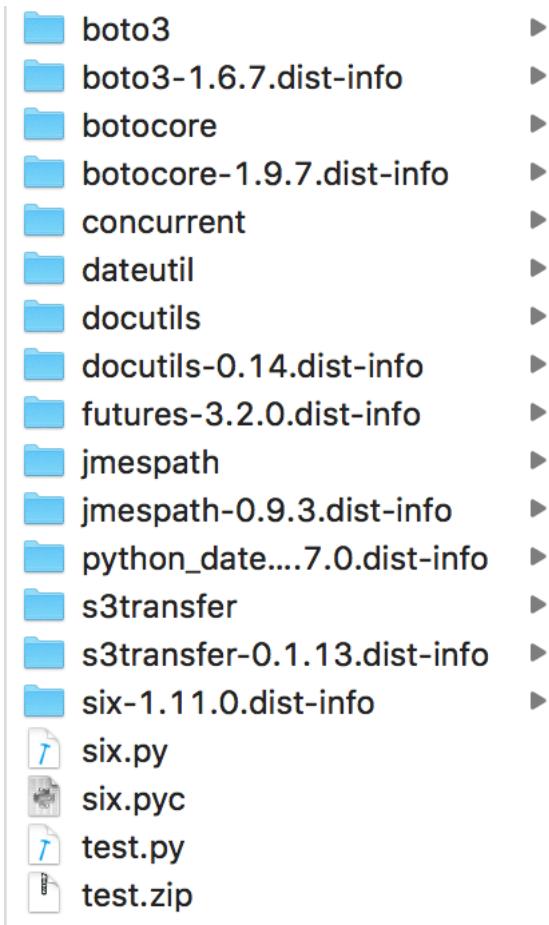


Figure 6 : Ensemble de dossiers et fichiers chargés dans AWS LAMBDA et nécessaire à l'exécution de notre fonction lambda dans le cloud Amazon

Nous avons par la suite compressé le tout en un fichier « test.zip » et l'avons déployé sur le cloud en suivant les étapes mentionnées plus haut dans la documentation section « création d'une fonction lambda simple ». Avant de déployer notre fonction à partir d'un fichier .zip, nous avons précisé le gestionnaire de la fonction et le langage utilisé. Nous avons ensuite testé le bon fonctionnement de notre fonction en créant un évènement de test montré sur la capture ci-dessous :

Configurer un événement de test

X

Une fonction peut comporter 10 événements de test maximum. Les événements sont conservés. Vous pouvez donc basculer vers un autre ordinateur ou navigateur web et tester votre fonction avec les mêmes événements.

- Créer un nouvel événement de test
- Modifier des événements de test enregistrés

Événement de test enregistré

▼
C

```

1 {
2   "value": 29,
3   "time": 1527647209
4 }
```

Figure 7: Événement de test configuré dans AWS LAMBDA afin de tester le bon fonctionnement de notre fonction lambda avec « value » comme données de température recueillies et « time » l'heure au format POSIX.

Après avoir configuré notre événement de test, nous l'avons exécuté et obtenu en sortie le journal d'exécution de notre fonction montré sur la capture ci-dessous :

⌚ Résultat de l'exécution : réussite ([journaux](#))

▼ Détails

La zone ci-dessous affiche le résultat renvoyé par l'exécution de votre fonction.

```
null
```

Récapitulatif

Code SHA-256	9fmNF5THcmOaiLLloJxkCRa0xn8i2wVn4Eq2hW9MXIM=	ID de demande	4f9dedcb-6920-11e8-8a0f-57d9a0778585
Durée	2085.04 ms	Durée facturée	2100 ms
Ressources configurées	128 MB	Mémoire max utilisée	33 MB

Sortie de journal

La zone ci-dessous affiche les appels de journalisation dans votre code. Ces derniers correspondent à une seule ligne du groupe de journaux CloudWatch correspondant à cette fonction Lambda. [Cliquez ici](#) pour afficher le groupe de journaux CloudWatch.

```

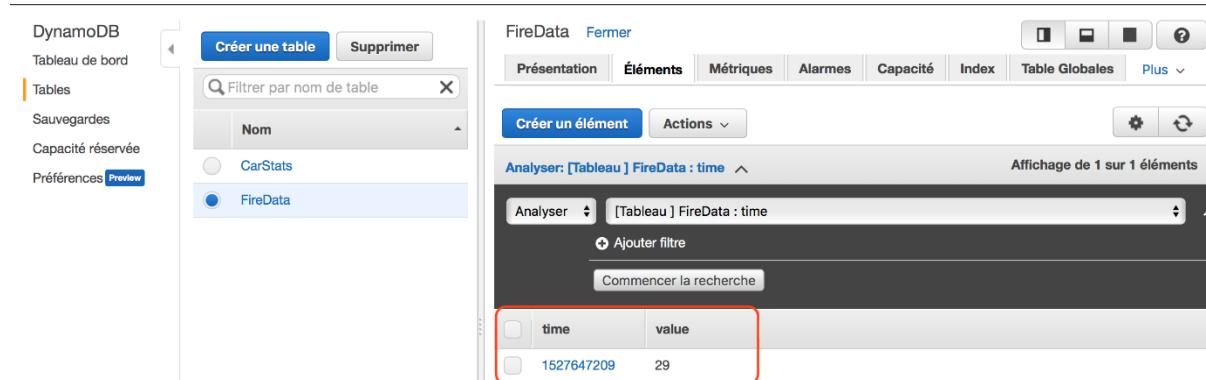
START RequestId: 4f9dedcb-6920-11e8-8a0f-57d9a0778585 Version: $LATEST
Table already created
[INFO] 2018-06-06T00:27:05.540Z 4f9dedcb-6920-11e8-8a0f-57d9a0778585 {u'value': 29, u'time': 1527647209}
[INFO] 2018-06-06T00:27:05.540Z 4f9dedcb-6920-11e8-8a0f-57d9a0778585 time state: 1527647209
[INFO] 2018-06-06T00:27:05.540Z 4f9dedcb-6920-11e8-8a0f-57d9a0778585 value state: 29
END RequestId: 4f9dedcb-6920-11e8-8a0f-57d9a0778585
REPORT RequestId: 4f9dedcb-6920-11e8-8a0f-57d9a0778585 Duration: 2085.04 ms Billed Duration: 2100 ms Memory Size: 128 MB Max Memory Used: 33 MB

```

Figure 8: Journal et résultat d'exécution de notre fonction lambda y compris la durée facturée par Amazon.

Nous observons que notre fonction s'est bien exécutée et nous remarquons dans la partie « Récapitulatif » le temps facturé pour l'exécution de notre fonction qui est de 2100 ms (ce temps est relativement long car lors du premier appel de la fonction, notre table de données « FireData » est créée, ce temps diminue considérablement lors des prochains appels de la fonction).

Nous pouvons remarquer sur la capture ci-dessous que nos données de test ont bien été écrites dans notre table « FireData » onglet « Elements » :



time	value
1527647209	29

Figure 9: Données de test écrites dans notre table "FireData" contenue dans la base de données AWS DYNAMODB.

Pour que tout cela soit possible, nous avons au préalable défini un rôle d'exécution qui permet de donner des droits d'accès à notre fonction lambda afin qu'elle puisse utiliser d'autres services. Nous avons parcouru la documentation [1.3.4] afin de configurer notre rôle IAM. La capture ci-dessous présente les rôles attribués à notre fonction lambda :

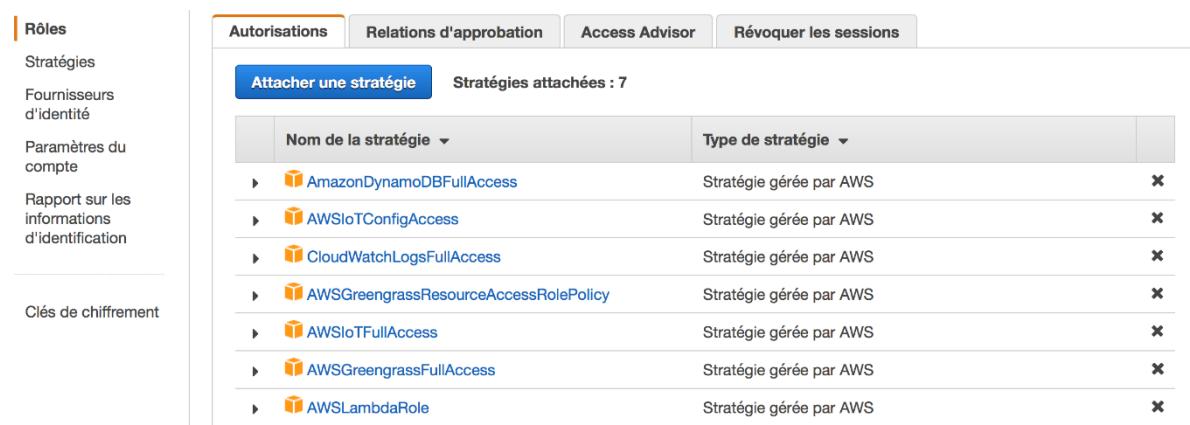


Figure 10: Rôles IAM (Identity and Access Management) attribués à notre fonction lambda afin qu'elle puisse accéder aux services AWS IOT, DYNAMODB, Greengrass et CloudWatch (journalisation) d'Amazon.

Ces différents rôles permettent de donner un accès total à notre fonction lambda aux services AWS DYNAMODB (Base de données dans le cloud), à la journalisation (CloudWatch), au service AWS IOT (Communication avec notre appareil) et AWS GREENGRASS (Pour le déploiement en local) que nous verrons plus en détail dans le chapitre IV.

III- CHOIX TECHNIQUES

Dans cette partie, nous allons décrire les différents composants de notre architecture et ainsi donner les raisons du choix de chacun.

1- Choix de l'équipement matériel

Pour pouvoir communiquer avec nos différents capteurs et traiter les données localement ou de les transmettre sur une base de données dans le cloud, nous nous sommes équipés d'une carte mère RASPBERRY PI 3 type B ayant les caractéristiques suivantes :

- Processeur intégré **Quad-Core ARM Cortex-A53 1.2 GHz**
- RAM : **1024**
- Wifi b/g/n et Bluetooth 4.1
- Basse consommation

Pour avoir plus de détails en ce qui concerne les équipements, nous avons consulté [1.4.1]. La caractéristique importante sur notre RASPBERRY PI est sa carte wifi qui peut faire office de passerelle pour les appareils IOT et le cloud. Nous avons installé comme système d'exploitation "Raspbian GNU/Linux 8 (jessie)".

Nous avons associer comme périphériques à notre RASPBERRY PI, une carte GROVE PI qui est un kit permettant la connexion de nos différents capteurs de température, d'humidité, de pression et de fumée ainsi qu'un écran tactile de cinq pouces sur lequel est affiché notre interface graphique. Cet écran tactile est doté d'un stylet ayant pour but de faciliter l'interaction avec les utilisateurs. Ci-dessous sont présentées les différentes images correspondant à notre RASPBERRY PI, à notre carte GROVE PI et à notre écran tactile.



Figure 11: RASPBERRY PI 3 modèle B [1.4.7]

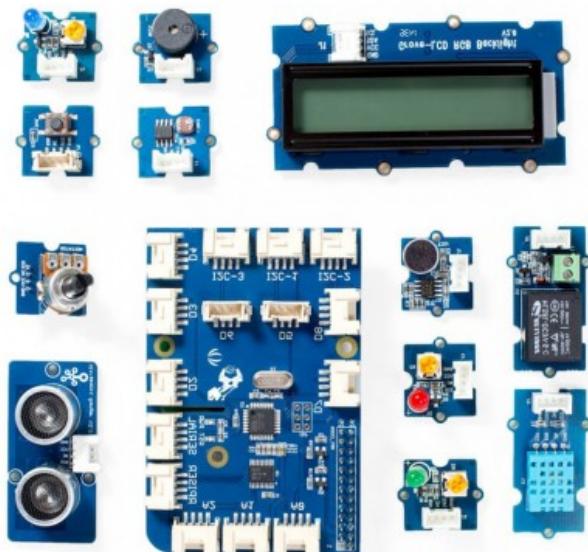


Figure 12: Kit de démarrage GROVE PI pour RASPBERRY PI [1.4.8]

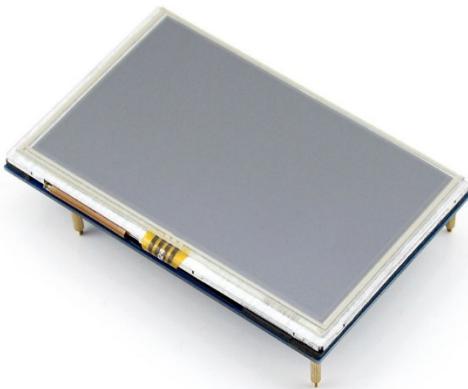
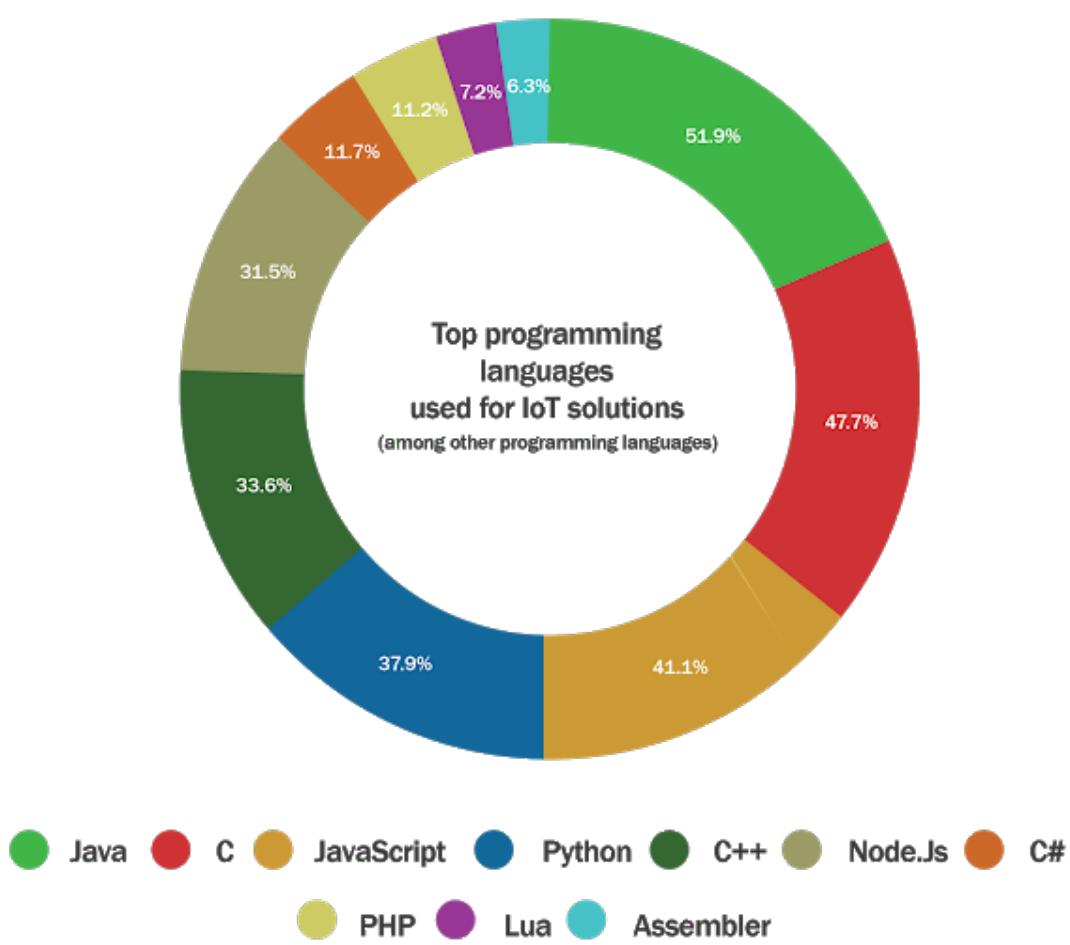


Figure 13: Écran tactile 5 pouces pour RASPBERRY PI [1.4.9]

2- Choix du langage de programmation

Étant donné que les langages supportés par AWS LAMBDA pour écrire le code de nos fonctions lambda sont : Node.js, Java, C#, Python et Go [1.4.2], nous avons choisi le langage python car il est le meilleur langage pour développer des scripts, il est une plateforme idéale pour la programmation réseaux. Il est également utilisé par une très grande communauté de développeurs.



(*Statistics - Eclipse IoT Working Group. IEEE IoT & AGILE IoT)

Figure 14: Diagramme comparatif des différents langages de programmation utilisé en IoT [1.4.3].

3- Choix de la base de données

On distingue deux types de base de données à savoir les bases de données SQL qui sont des bases de données relationnelles utilisant un modèle de données relationnelles sous forme de schéma relationnel, et les bases de données NoSQL qui n'utilisent pas un modèle de données relationnel mais utilisent plusieurs modèles de données non relationnels (clé-valeur, orienté document, famille de colonne et orienté graphe).

Nous avons premièrement utilisé la base de données DYNAMODB fourni par Amazon pour réaliser la phase d'expérimentation de la fonction lambda réalisée dans le chapitre II. Cependant, Amazon ne permet pas d'installer DYNAMODB (pas OpenSource) en local sur le RASPBERRY PI, raison pour laquelle nous nous sommes penchés sur une autre solution de base de données noSQL : Couchdatabase version 2.1.1. Certes elle n'est pas la plus utilisée dans le monde professionnel comme on peut l'observer sur le graphe ci-dessous mais elle

s'adapte facilement aux variations de trafic, aux grands volumes de données, elle est tolérante aux pannes et facile d'utilisation [1.4.4].

Un tableau comparatif entre couchdb et mongodb [1.4.5] permet de mieux distinguer la différence entre ces deux bases de données. Cependant, cette comparaison ne reflète ni la fiabilité ni la robustesse des différentes bases de données mais la popularité de chacune de ces bases sur internet à savoir les blogs, les forums, les tutoriels et les publicités, leur utilisation par les entreprises.

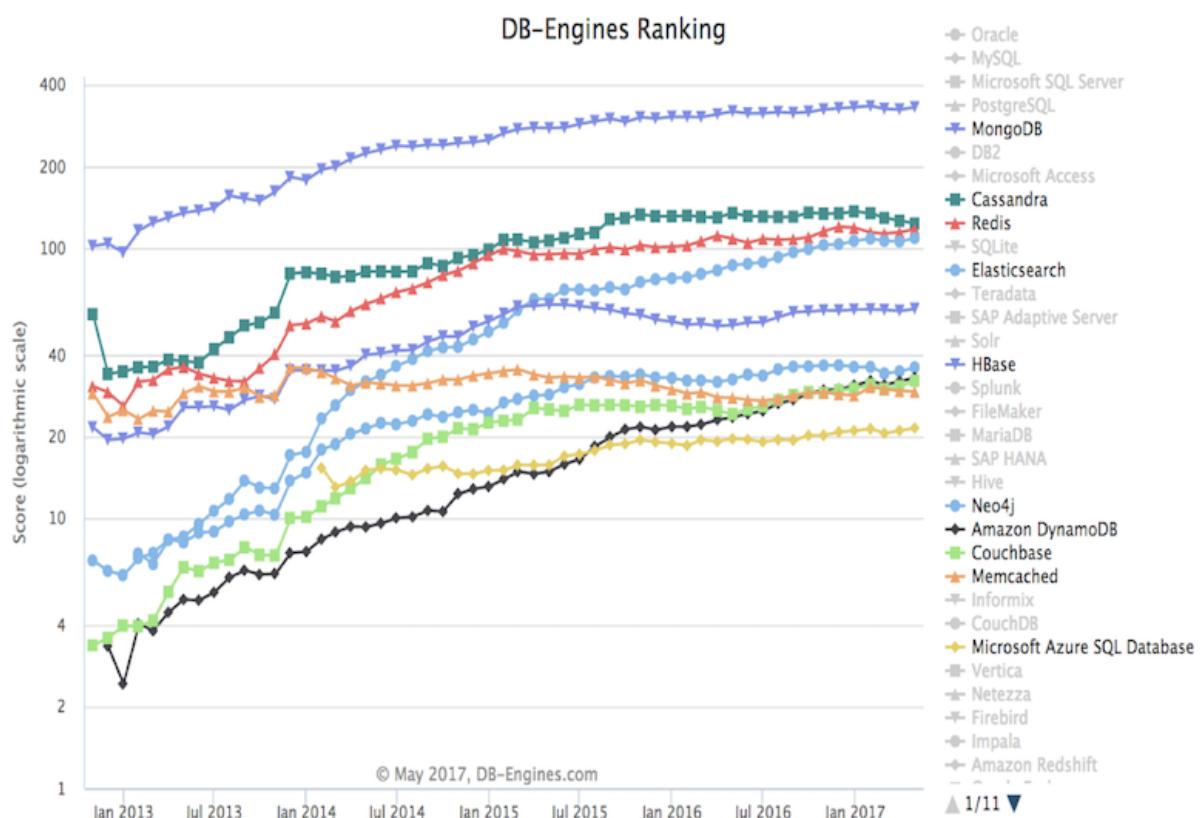


Figure 15: Graphe comparatif des différentes bases de données NoSQL [1.4.6].

IV- CONCEPT EDGE COMPUTING

Dans l'optique de déployer le traitement et la sauvegarde de nos données en local, nous avons choisi le service AWS GREENGRASS fourni par AWS IOT. Pour réaliser ce déploiement, nous allons configurer une passerelle (noyau greengrass sur lequel s'exécute notre fonction lambda), un agent de messages utilisant le protocole MQTT, un service de sécurité et d'identité, un shadow d'appareil et un service Device Shadow nécessaire pour la réalisation de la partie affichage.

Nous allons dans un premier temps détailler les différents éléments nécessaires à la réalisation du projet et dans un deuxième temps, avec l'utilisation de la plateforme AWS IOT, déployer la sauvegarde et le traitement de nos données en local.

1- Composants AWS IOT

Dans cette partie nous décrivons les différents outils nécessaires à la réalisation de notre projet afin de déployer la récupération, la sauvegarde et le traitement des données en local.

a- Passerelle machine to machine

Afin de pouvoir acheminer nos données lues par les capteurs vers le cloud AWS ou vers notre base de données de manière sécurisée et efficace, nous avons besoin d'une passerelle.

Dans notre cas il s'agit tout simplement d'un RASPBERRY PI sur lequel est installé un noyau greengrass qui se connecte au cloud AWS afin d'envoyer les informations reçues dans le cloud et sur lequel est enregistré nos différents appareils. Ensuite notre fonction lambda est déployée dans ce noyau afin de récupérer les données de capteurs et de les sauvegarder dans une base de données en local.

b- Agent de messages

Nos appareils utilisent le protocole MQTT sur le port 8883 pour communiquer avec notre passerelle et le cloud AWS de manière sécurisée. Ce service permet la publication de messages ou l'abonnement sur une rubrique.

c- Service de sécurité et d'identité

Le trafic entre nos appareils et les autres services AWS IOT à savoir l'agent de message, le service Device Shadow, est chiffré sur le protocole TLS (Transport Layer Security). Ce protocole garantit la confidentialité des autres protocoles utilisés par AWS IOT à savoir le protocole MQTT et http. C'est pour cette raison qu'une identification est nécessaire au préalable et cela est rendu possible grâce à l'utilisation des clés publiques et privées ainsi que des certificats qui sont générés lors de l'enregistrement d'un appareil sur la plateforme AWS IOT.

d- Shadow d'appareil

Ce shadow d'appareil est en effet un document JSON qui permet le stockage des informations concernant l'état d'un appareil. Ce document permet à plusieurs appareils d'échanger leurs états dans le but de se synchroniser en local ou dans le cloud. Il peut aussi être accessible et modifié depuis une application distante.

e- Service Device Shadow

Dans le but d'avoir une représentation de nos appareils dans le cloud afin de synchroniser les états mis à jour par nos appareils avec d'autres appareils ou applications lors de la connexion de ceux-ci, AWS IOT met à disposition ce service.

En effet lorsqu'un appareil publie son état dans le device shadow, cet état correspond à un état reporté (actuel) de l'appareil. Si cet état doit être mis à jour par un utilisateur ou une application, alors cet état de mis à jour correspond à un état désiré.

Le changement d'état de « reporté » à « désiré » s'effectue sur la rubrique « \$aws/things/ThingName/shadow/delta ». La publication d'un document s'effectue sur la rubrique « \$aws/things/ThingName/shadow/update ». Pour obtenir plus de détails sur les rubriques de publication et d'abonnement, veuillez consulter [1.5.3].

Ce service permet la gestion des versions pour chaque message de mis à jour (incrémentation de la version par message). Dans le cas où plusieurs versions sont reçues par notre device shadow, il sélectionne uniquement la version récente et supprime les anciennes. Car l'ordre d'arrivée des messages n'est pas garanti par le service.

2- Stockage des données en local

Après avoir compris comment effectuer la sauvegarde de nos données sur le cloud grâce aux fonctions lambda, nous avons réalisé cette sauvegarde en local. Car une rupture de connexion réseau avec le cloud entraîne une perte de données. Les étapes qui vont suivre montrent comment nous avons procédé afin de décentraliser notre traitement en local sur notre RASPBERRY PI.

Nous avons suivi les différentes étapes proposées dans la documentation d'Amazon [1.5.1]. AWS GREENGRASS permet le déploiement et l'exécution en local de nos fonctions lambda. Ces fonctions répondent à des événements générés par des messages sur une rubrique provenant soit du cloud, soit d'autres appareils (objets interconnectés abonnés à la rubrique). Comme nous l'avons déjà expliqué, les appareils IoT communiquent avec notre noyau Greengrass en local via le protocole MQTT. Toutes les informations de sécurité et de configurations sont définies dans un groupe AWS GREENGRASS dans le but de paramétrier les équipements connectés dans notre réseau local. Nous avons de ce fait défini un groupe nommé « RaspiGroup2 » dans lequel nous avons configuré des abonnements, défini les fonctions lambda à déployer en local, les rôles et ajouté des appareils AWS IOT qui seront connectés à notre noyau.

Dans le but d'avoir une redondance de nos données, nous avons décidé d'utiliser le mécanisme de réPLICATION des données qui consiste à créer une copie d'une base de données sur un serveur vers un autre serveur distant. Il existe deux modèles de réPLICATION à savoir « Master – Slave » et « Peer-to-Peer ».

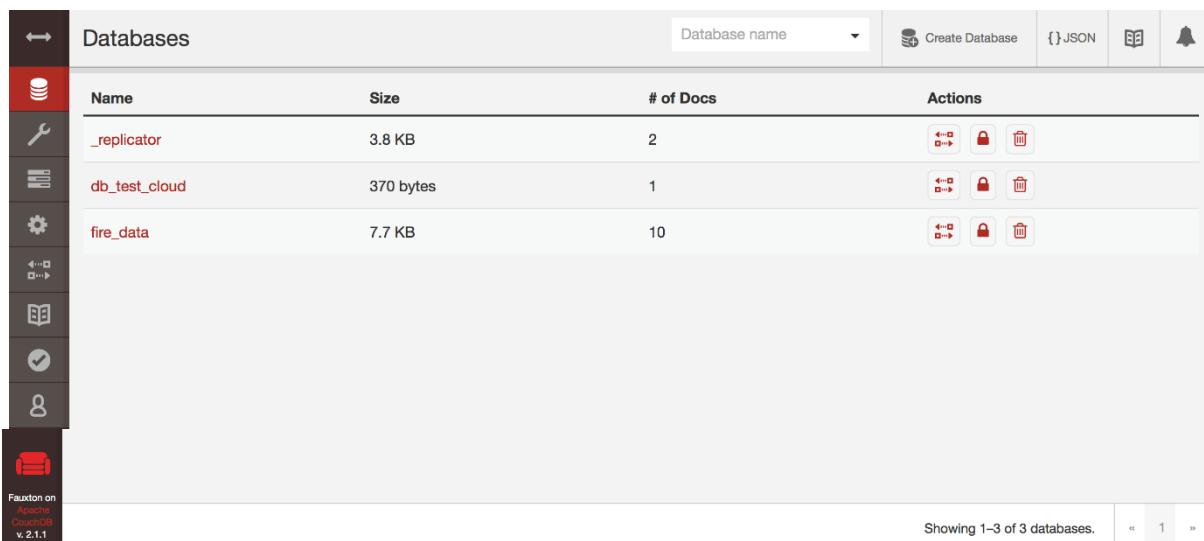
S’agissant de la réPLICATION « Master-Slave », les opérations d’écriture et de lecture s’effectuent sur le serveur « Master » tandis qu’uniquement des opérations de lecture s’effectuent sur le serveur « Slave ».

Quant à la réPLICATION « Peer-to-Peer », les opérations d’écriture et de lecture s’effectuent sur tous les serveurs. En d’autres termes si une écriture est effectuée, elle sera répliquée sur tous les autres serveurs.

Nous avons installé la base de données couchdb version 2.1.1 sur notre RASPBERRY PI et sur une instance EC2 sur le cloud afin d’avoir une même copie de nos données en local et dans le cloud. Nous avons choisi le modèle de réPLICATION « Master-Slave » Car nous effectuons des écritures et lectures sur la base de données en local qui correspond à notre serveur « Master ». Nous avons de ce fait configuré notre base de données couchDB en local afin qu’elle puisse communiquer avec celle dans le cloud.

Cette configuration nécessite de préciser une source, une destination et le type de réPLICATION (de manière continue ou juste une fois). Nous avons choisi le type de réPLICATION continu dans l’optique de ne répliquer qu’une faible quantité de données et de ne pas perdre des données tandis que l’autre option effectue une copie complète de la base de données à un instant donné entraînant ainsi une perte de données et de temps.

La capture ci-dessous correspond à l’interface graphique de notre base de données en local.



Name	Size	# of Docs	Actions
_replicator	3.8 KB	2	
db_test_cloud	370 bytes	1	
fire_data	7.7 KB	10	

Fauxton on Apache CouchDB v.2.1.1

Showing 1–3 of 3 databases. « 1 »

Figure 16: Interface graphique Fauxton de CouchDB montrant que les données ont bien été inscrites dans notre table de données fire_data.

Nous avons installé un système Ubuntu 16.04 sur notre instance EC2 afin de disposer de tous les packages lors de l’installation de notre base de données. La capture ci-dessous nous montre les caractéristiques de notre instance à savoir le type d’instance, l’état et les caractéristiques réseaux (DNS, adresse IP public et privée).

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)
Couchdb_test	i-057cf55c786de5b7e	t2.micro	eu-central-1b	running	2/2 checks ...	None	ec2-35-158-172-54
Instance: i-057cf55c786de5b7e (Couchdb_test)		Public DNS: ec2-35-158-172-54.eu-central-1.compute.amazonaws.com					
Description	Status Checks	Monitoring	Tags				
Instance ID	i-057cf55c786de5b7e			Public DNS (IPv4)	ec2-35-158-172-54.eu-central-1.compute.amazonaws.com		
Instance state	running			IPv4 Public IP	35.158.172.54		
Instance type	t2.micro			IPv6 IPs	-		
Elastic IPs				Private DNS	ip-172-31-39-74.eu-central-1.compute.internal		
Availability zone	eu-central-1b			Private IPs	172.31.39.74		
Security groups	Serverless_TB, view inbound rules			Secondary private IPs			
Scheduled events	No scheduled events			VPC ID	vpc-60309a0b		
AMI ID	ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180306 (ami-7c412f13)			Subnet ID	subnet-0e70db73		

Figure 17: Description de l'instance EC2 sur laquelle est installé couchDB dans le cloud d'Amazon

Nous avons ensuite configuré le pare feu de notre instance afin d'accepter les communications entrantes avec notre base de données en local, avec nos requêtes http depuis notre navigateur et d'éventuels « Ping » pour des tests de connexion.

N.B : Le service AWS DYNAMODB ne peut pas être géré en local mais uniquement dans le cloud d'où l'utilisation de couchdb.

Étant donné que le service AWS LAMBDA d'Amazon ne dispose pas de la librairie couchdb nécessaire pour établir la connexion et interagir avec la base de données et notre fonction lambda, nous avons téléchargé la librairie via le lien suivant [1.5.2] et avons chargé le fichier « .zip » comme vu dans le chapitre II dans la console AWS LAMBDA.

La capture ci-dessous montre le fichier compressé et son contenu intitulé « test.zip ». Le dossier intitulé « couchdb » importé dans notre code permet de nous connecter avec notre base de données couchdb en local. Nous avons importé ce package car il n'est pas disponible sur l'instance Amazon dans laquelle est exécutée notre fonction lambda.

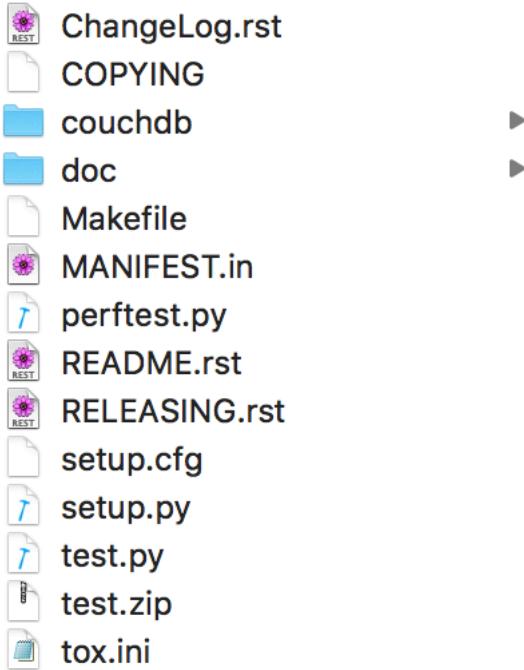


Figure 18: Ensemble de dossiers et fichiers chargés dans AWS LAMBDA et nécessaire à l'exécution de notre fonction lambda en local

Après avoir installé notre base de données couchdb en local et sur notre instance EC2 dans le cloud, nous avons modifié notre fonction lambda afin qu'elle puisse écrire directement les données reçues des capteurs dans la base de données en local. La capture ci-dessous correspond à notre code modifié.



```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import logging
5 import couchdb
6
7
8 # Connexion à la base de donnée
9 couch = couchdb.Server('http://raspberrypi.home:5985')
10
11
12 # nous vérifions si notre base de donnée existe dans le cas contraire nous la créons
13 dbname = "fire_data"
14 if dbname in couch:
15     print('Database already exist')
16     # Sélection de notre base de donnée
17     db = couch[dbname]
18 else:
19     # Création de notre base de donnée
20     db = couch.create('fire_data')
21     print('Database created')
22
23
24 # Initialisation du logger
25 logger = logging.getLogger()
26 logger.setLevel(logging.INFO)
27
28
29 def function_handler(event, context):
30
31     # récupération de l'événement et extraction des valeurs
32     # schema JSON :
33     # { "time":DateTime, "value":number}
34     logger.info(event)
35     time = event["time"]
36     value = event["value"]
37     logger.info("time state: " + str(time))
38     logger.info("value state: " + str(value))
39
40     # objet JSON qui sera sauvegarder dans la base de donnée couchdb
41     Item={
42         'time':time,
43         'value':value
44     }
45
```

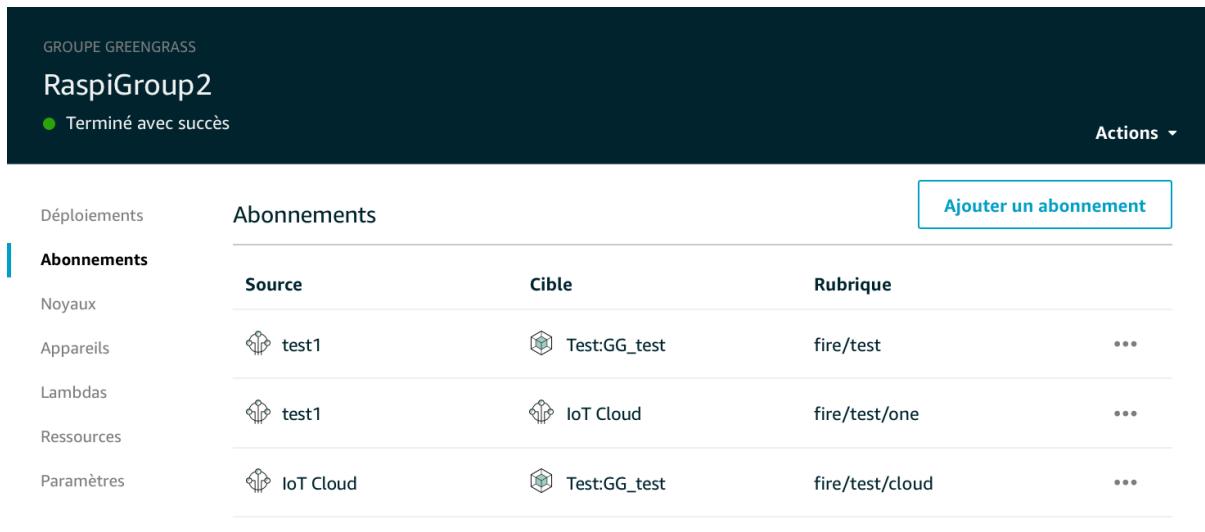
Figure 19: Fonctions lambda permettant la connexion et l'écriture des données en local qui est déployée sur notre noyau AWS GREENGRASS.

Après avoir modifié et déployé en local notre fonction lambda, nous avons généré des données aléatoires d'un capteur suivant le format JSON ci-après {‘time’ : ‘timestamp au format POSIX’, ‘value’ : nombre aléatoire compris entre 15 et 30 degré Celsius}.

Les communications entre notre capteur (données simulées via un script) et notre RASPBERRY PI (qui joue office de passerelle) s'effectue via le protocole MQTT (Message Queuing Telemetry Transport) basé sur TCP/IP, qui permet l'échange de message entre un « publisher » (source) et un « subscriber » (cible) abonnés à un « topic » (rubrique).

La communication entre nos équipements est sécurisée par l'utilisation des certificats X.509, des stratégies et des rôles IAM.

Nous avons ajouté plusieurs abonnements dans la console AWS IOT comme montré sur la capture ci-dessous afin de gérer la communication entre notre appareil (correspondant pour le moment à notre script python) et notre fonction lambda :



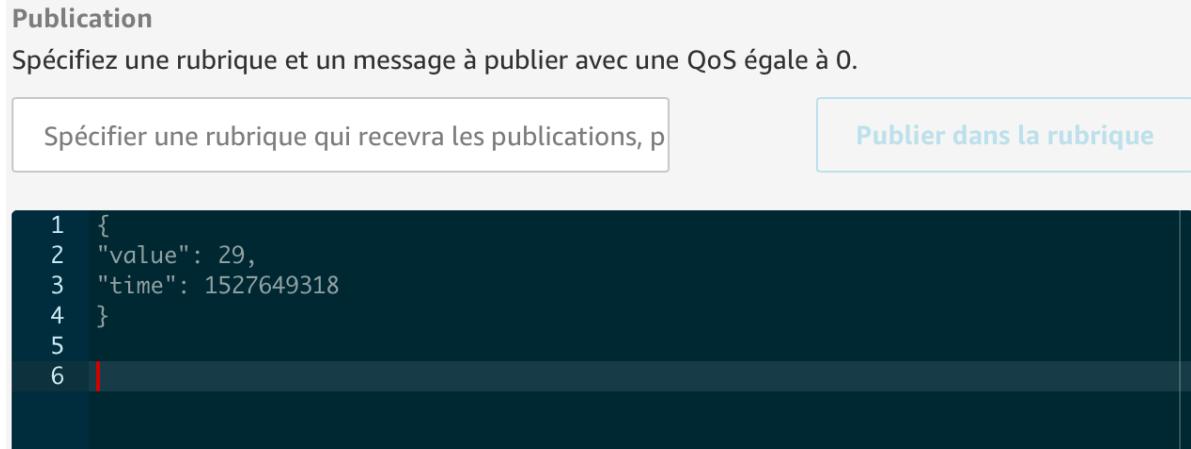
The screenshot shows the AWS Greengrass group 'RaspiGroup2'. It has three subscriptions listed:

Source	Cible	Rubrique
test1	Test:GG_test	fire/test
test1	IoT Cloud	fire/test/one
IoT Cloud	Test:GG_test	fire/test/cloud

Figure 20: Abonnements gérant la communication entre les différentes sources et cible en fonction de chaque rubrique.

Nous observons sur la capture ci-dessus que la source (publisher) peut être soit notre appareil (capteur), soit le cloud IoT. La cible (subscriber) correspondant au cloud AWS IoT et à notre fonction lambda ayant pour alias GG_test, chacun communiquant sur une rubrique différente. Nous avons ensuite générée les données de deux manières distinctes :

- Depuis le cloud AWS IoT en utilisant l'option « test » fourni dans la console AWS IOT comme montré sur la capture ci-dessous :



The screenshot shows the AWS IoT Publish interface. A JSON message is being published to the topic 'fire/test/cloud':

```

1  {
2    "value": 29,
3    "time": 1527649318
4  }
5
6

```

Figure 21: Génération d'un événement depuis la console AWS IOT dans le cloud afin de tester le fonctionnement de notre fonction lambda déployé en local.

- Depuis notre script simulant les données envoyées par un capteur de température, dont le bout de code (ce code correspond à celui de TrafficLight.py contenu dans la phase d'entraînement de la section « Démarrer avec AWS GREENGRASS et à juste été modifié afin de générer des données de capteurs) est présenté sur la capture ci-dessous :

```
# there we add our modification to generate our random values in JSON format
while True:
    if args.mode == 'publish':
        message = {}
        date = datetime.datetime.now()
        # message['time'] = datetime.datetime.today().replace(microsecond = 0).isoformat(' ')
        message['time'] = calendar.timegm(date.timetuple())
        # generate random temperature value in a house
        message['value'] = randint(15,35)
        messageJson = json.dumps(message)
        myAWSIoTMQTTClient.publish(topic, messageJson, 0)
        if args.mode == 'publish':
            print('Published on topic %s: %s\n' % (topic, messageJson))
        time.sleep(10)
```

Figure 22: Bout de code simulant un appareil envoyant au format JSON des données de températures à un moment donné (sous format POSIX) sur la rubrique « fire/test »

Ci-dessous, nous observons effectivement que nos données ont été écrites dans notre base de données couchdb.

```
1 {  
2     "_id": "580ee02a3dfe909c93e5c12acc002545",  
3     "_rev": "1-e2a5523cddd4ab29e787d8cf302790cd",  
4     "value": 24,  
5     "time": 1527649318  
6 }
```

Figure 23: Données écrites dans notre base de données référencées par un `_id` qui est un identifiant unique représentant le nom du document et `_rev` (revision) utilisé pour le contrôle de la concurrence

Partie II : Conception et réalisation

Dans cette partie nous réalisons l'architecture de simulation de notre projet (prototype) en incluant chaque composant nécessaire à sa compréhension, ensuite nous concevons notre architecture depuis les capteurs jusqu'à la vue qui correspond à l'interface utilisateur en local et dans le cloud.

I- Architecture détaillée de notre infrastructure

Afin d'avoir une idée de notre architecture, nous avons réalisé les schémas ci-dessous afin d'apporter une meilleure compréhension du flux de données et le rôle de chaque programme dans la réalisation de notre simulation.

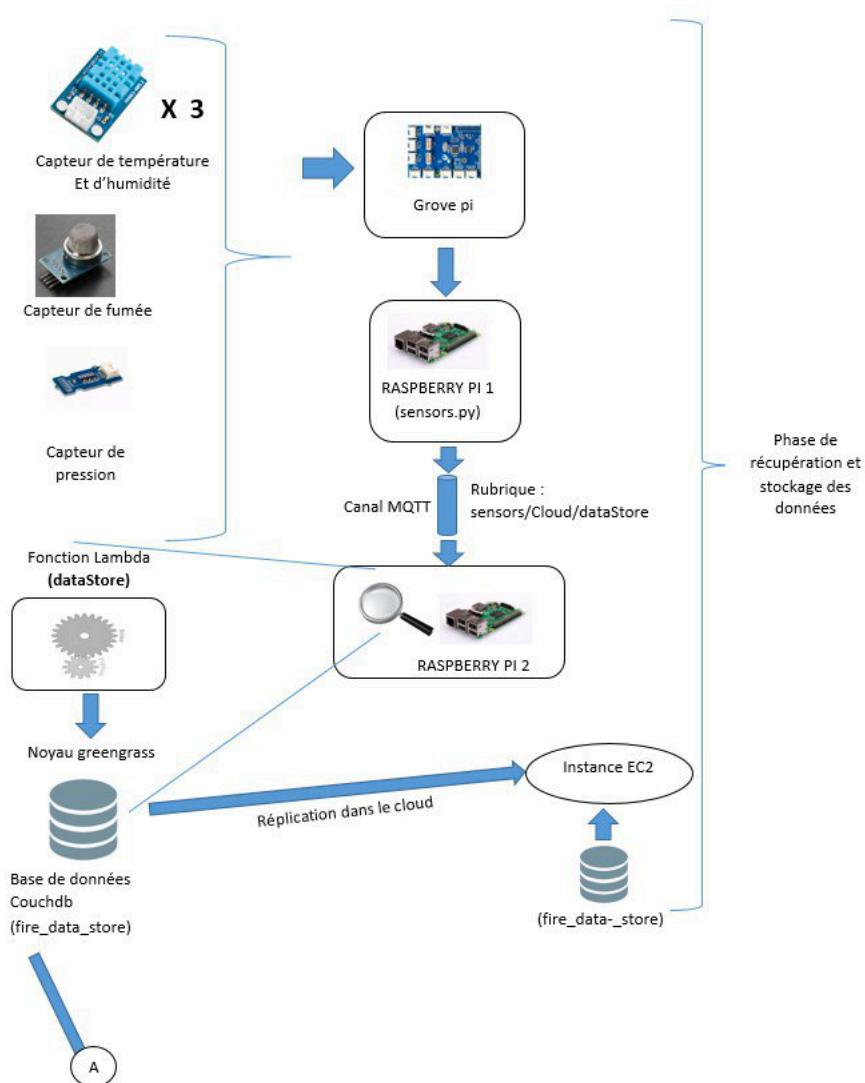


Figure 24: Parcours détaillé de nos données depuis les capteurs à la base de données. Ce parcours résume la phase de récupération et de stockage des données.

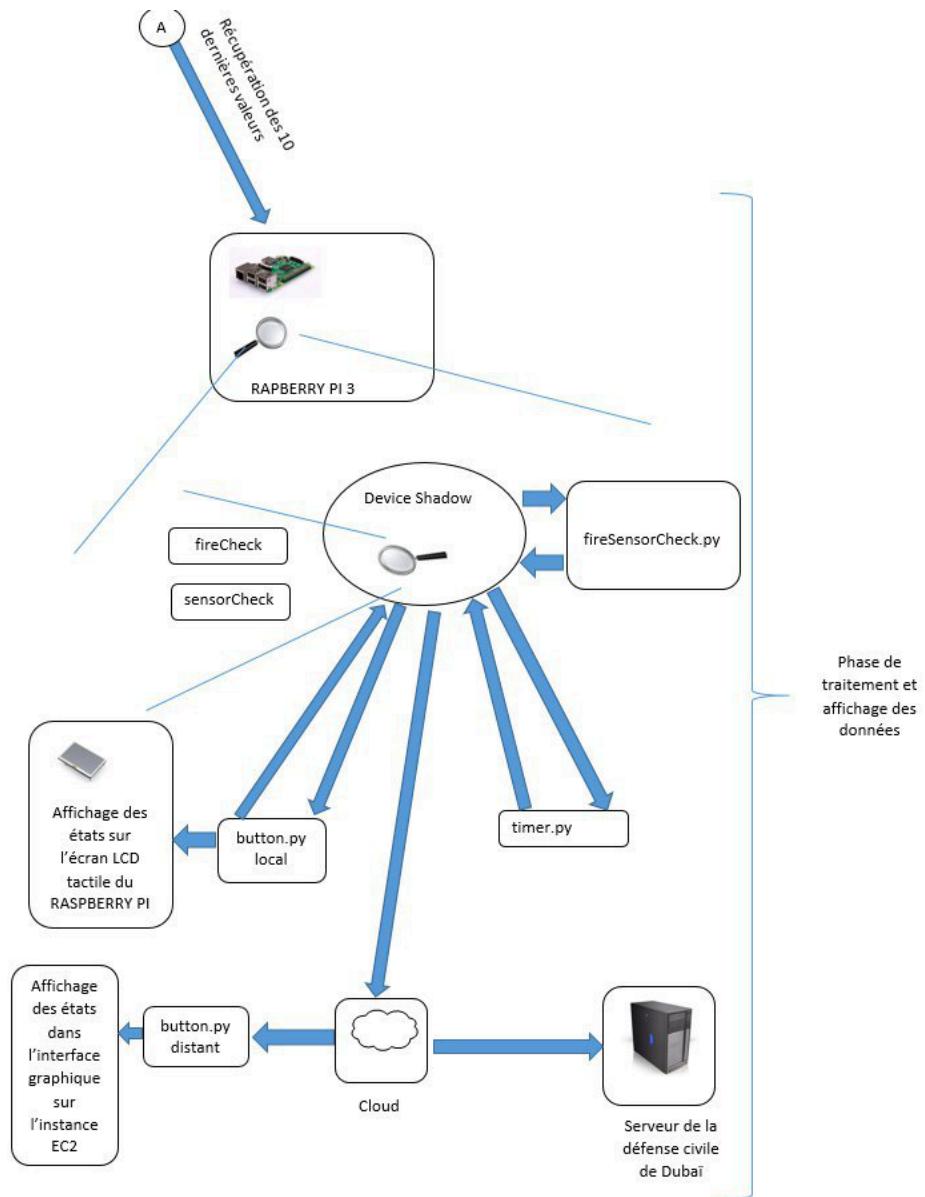


Figure 25: Parcours détaillé de nos données depuis la base de données à l'interface graphique et au serveur de la défense civile de Dubaï. Ce parcours résume la phase de traitement et d'affichage des données.

Pour mieux expliquer le parcours détaillé de nos données, nous avons divisé celui-ci en deux phases à savoir : une phase de récupération et de stockage des données et une autre phase de traitement et affichage de ces données.

1- Phase de récupération et de stockage des données

Cette partie a pour but d'expliquer le processus de lecture des données par nos capteurs ainsi que l'écriture de celles-ci dans notre base de données.

La récupération de nos données s'effectue par une série de capteurs dont le but est de collecter les données de température, d'humidité, de fumée et de pression dans une villa comme montré dans notre architecture générale. Ces capteurs sont reliés à une carte GROVE PI qui joue le rôle d'interface entre nos capteurs et notre RASPBERRY PI.

Les données générées par nos capteurs sont ensuite récupérées et transmises sur la rubrique « sensors/cloud/dataStore » dans un payload contenant le document JSON avec nos valeurs. Ces opérations sont effectuées par notre script python (sensors.py). Pour pouvoir communiquer avec le service AWS IOT chaque appareil doit au préalable s'authentifier et se connecter au serveur AWS IOT plus précisément à notre mini cloud.

Cette authentification s'effectue dans notre cas grâce à des clés publiques et privées générées dans la console AWS IOT lors de la création et la configuration de notre appareil, ainsi que des certificats normalisés au standard X.509 qui ont une longue durée de vie (jusqu'au 31 décembre 2049 horaire GMT). Ces certificats permettent à nos appareils de garantir qu'ils communiquent avec le serveur AWS IOT et non un autre. Une fois authentifié et connecté avec AWS IOT, l'appareil publie sur la rubrique « sensors/cloud/dataStore » des informations.

Ce canal est du type MQTT (Message Queuing Telemetry Transport) qui est un protocole de messagerie publish-subscribe basé sur le protocole TCP/IP [2.1.1]. Ensuite nous récupérons les données générées par les capteurs et nous les publions sur la rubrique MQTT mentionnée plus haut au format JSON.

Ces données sont ensuite reçues par notre fonction lambda, déployées dans notre noyau greengrass en local et écrites dans notre base de données fire_data_store. Une fois les données écrites dans notre base de données en local, elles sont répliquées dans une base de données (fire_data_store) distante dans le cloud sur une instance EC2.

2- Phase de traitement des données

Cette partie explique le processus de traitement de nos données récupérées depuis notre base de données en local.

Afin d'effectuer ce traitement, un ensemble de données correspondant aux 10 dernières valeurs de chaque capteur est récupéré dans la base de données fire_data_store, à partir d'un script python (fireSensorCheck.py) localisé dans notre RASPBERRY PI. Ces données sont analysées afin de détecter si un incendie s'est déclenché ou pas et dans un deuxième cas de figure si un capteur ou plusieurs capteurs sont défaillants.

Afin d'échanger et de mettre à jour les différents états concernant, soit la détection d'incendie ou de capteurs défaillants, nous utilisons le service Device Shadow qui nous permet comme

montré dans l'architecture de l'infrastructure, de mettre à jour les états de chaque appareil depuis un autre appareil, une application locale ou distante et depuis le cloud AWS.

Nous observons deux « Device shadows » à savoir « fireCheck » et « sensorCheck » enregistrés dans la console AWS IOT. Ces deux « Device shadows » sont une représentation de notre appareil dans le cloud et afin d'afficher son état et d'être mis à jour. Des « Device shadows » différents sont enregistrés pour chaque villa afin d'empêcher les conflits lors de la mise à jour des états dans le local shadow. Nous avons deux device shadow par villa à savoir fireCheck (pour la détection d'incendie) et sensorCheck (pour la détection de capteurs défaillants).

a- Détection incendie

Dans le cas d'une détection d'incendie dans une villa, on dispose de plusieurs cas de figure : Si un capteur détecte un incendie et si plusieurs capteurs détectent simultanément un incendie.

Si un incendie est détecté par un capteur, due à des valeurs très haute de température ou de fumée, une alarme est émise et affichée sur l'interface graphique générée par notre script python (bouton.py) et un temporisateur de trente secondes est démarré. Pendant le décompte, l'utilisateur peut soit quittancer l'alarme soit la confirmer.

Si elle est confirmée, alors l'alarme est directement émise vers le serveur de la défense civil de Dubaï, par contre si elle est quittancée alors l'alarme est supprimée. Cependant si le décompte se termine et qu'aucune modification n'a été reçue de la part de l'utilisateur ou des données traitées, alors l'alarme est directement transmise sur les serveurs de la défense civile de Dubaï.

Si un incendie est détecté par plusieurs capteurs simultanément, alors l'alarme est directement transmise sur les serveurs de la défense civile de Dubaï.

Cette transmission d'alarme est gérée par des états qui sont mis à jour dans un « Device shadow ». La transition entre ces états est montrée sur la figure ci-dessous.

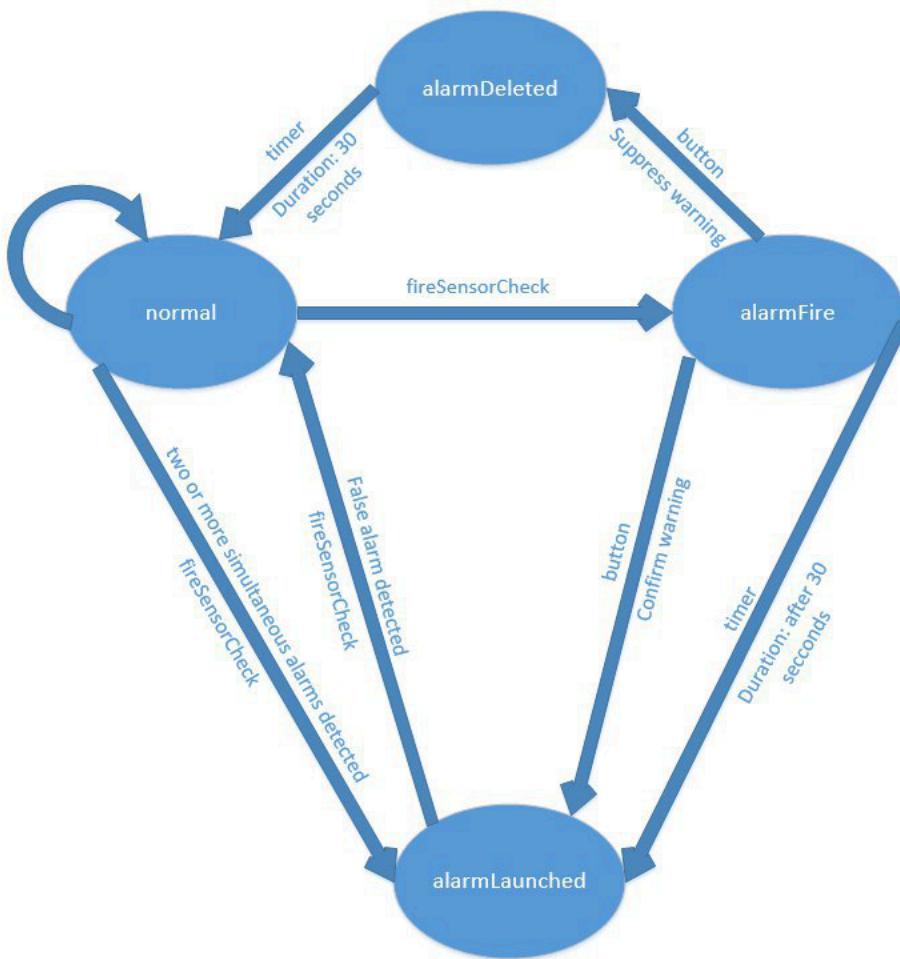


Figure 26: Diagramme d'états dans notre Device shadow "fireCheck"

Le fonctionnement des états est le suivant pour chaque villa :

Dans un premier temps notre script « fireSensorCheck.py » récupère et analyse les données de capteurs de température et de fumée. Nous disposons de trois capteurs de température et un capteur de fumée dont les valeurs sont traitées. Si après analyse un incendie est détecté alors l'état est mis à jour de « normal » à « alarmFire », si par contre aucun incendie est détecté alors on reste à l'état « normal ». Si un incendie est détecté par plusieurs capteurs, dans ce cas l'état change de « normal » à « alarmLaunched ».

S'il s'agit d'une fausse alarme due à une variation dans les données alors l'état passe de « alarmLaunched » à « normal ». Nous n'avons pas géré le retour à l'état « normal » par un bouton mais cela se produit par une variation des données traitées.

Une fois à l'état « alarmFire », un temporisateur de trente secondes est démarré par le script « timer.py » afin de permettre à un utilisateur de mettre à jour l'état dans le « Device shadow ».

Si l'alarme est confirmée par un utilisateur alors l'état change de « alarmFire » à « alarmLaunched », par contre si aucune mise à jour n'est effectuée par un utilisateur et que les trente secondes sont écoulées alors l'état change de « alarmFire » à « alarmLaunched »,

Si l'alarme est supprimée par un utilisateur, alors l'état change de « alarmFire » à « alarmDeleted ». Une fois à cet état, un temporisateur de trente secondes est démarré par le script « timer.py » afin que l'action de l'utilisateur ne soit pas modifiée « fireSensorcheck.py ». Une fois les trente secondes écoulées, nous changeons l'état de « alarmDeleted » à « normal ».

b- Détection d'un ou plusieurs capteurs défaillants

Dans le cas d'une détection de capteurs défaillants dans une villa, un temporisateur d'une minute est démarré afin de s'assurer que ces capteurs n'émettent plus de données. Si c'est le cas alors la liste des capteurs défaillants est affichée avec la localisation de chacun d'eux dans la villa sur l'interface graphique, afin que l'utilisateur puisse agir en conséquence tout en sachant exactement où se situe chacun des capteurs.

II- Réalisation de la partie capteurs

Il est question dans cette partie d'expliquer le processus de lecture des données générées par nos capteurs et de l'envoi de celles-ci à notre passerelle. Le graphe ci-dessous présente les grandes étapes effectuées par notre script afin de publier les données lues sur la rubrique.

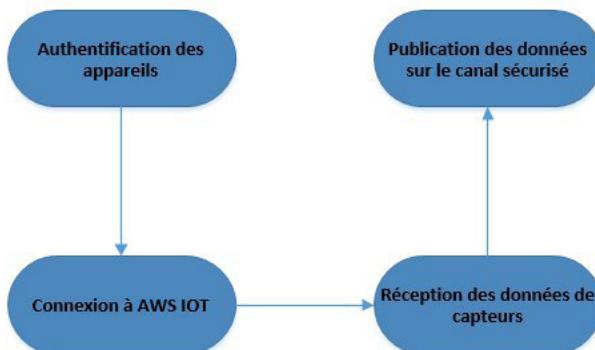


Figure 27: Diagramme d'état montrant le fonctionnement de notre script sensors.py

Pour réaliser cette partie nous avons utilisé un script python qui réalise les étapes suivantes :

- Afin de chiffrer la communication entre nos appareils, notre passerelle et notre agent de messages, nous authentifions nos appareils à partir des clés publiques et privées ainsi qu'avec le certificat délivré par Amazon. Ces clés et certificats sont stockés dans le même dossier que celui du script. Nous utilisons un script Bash afin de récupérer ces fichiers sous forme d'arguments et nous récupérons ces arguments dans notre script sensors.py.
- Après avoir connecté et authentifié nos appareils, nous attribuons à chaque capteur un UUID (Universal Unique Identifier) afin de pouvoir identifier chaque capteur de

manière unique. Ensuite nous effectuons une lecture des données recueillies par chaque capteur et nous les stockons dans un document JSON. Ce document est converti au format String avant d'être publié sur la rubrique « sensors/cloud/dataStore ». Les données de température sont exprimées en degré Celsius et celles de fumée, pression et humidité sont exprimées en pourcentage.

Nous avons affecté à nos capteurs des positions dans une villa comme montré sur le schéma ci-dessous.

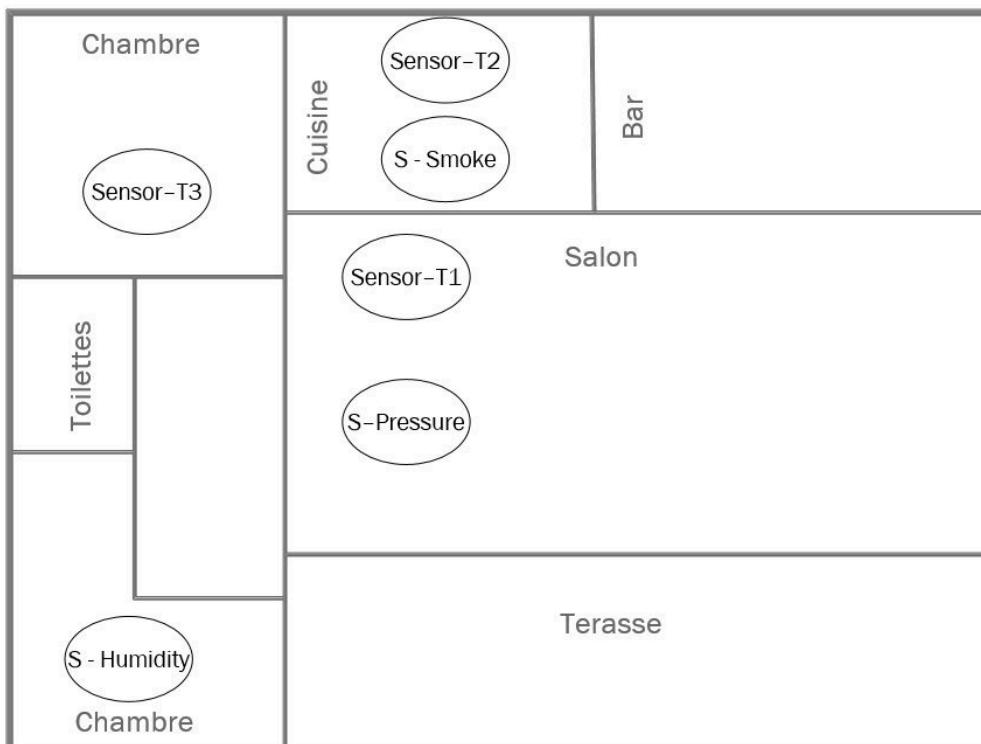


Figure 28: Position de nos capteurs dans une villa dans lequel les capteurs de températures sont représentés par Sensor-T1 à T3, le capteur d'humidité par S-Humidity, le capteur de pression par S-Pressure et le capteur de fumée par S-Smoke.

III- Réalisation de la partie stockage des données via la fonction lambda et réPLICATION dans le cloud

Le but de cette partie est d'implémenter une fonction lambda portant le nom de « dataStore », qui nous permettra de stocker les valeurs reçues des différents capteurs dans notre base de données en local « fire_data_store ». Pour mieux comprendre cette partie, nous avons schématiser le processus de fonctionnement de notre fonction lambda sous forme de diagramme représenté sur la figure ci-dessous :

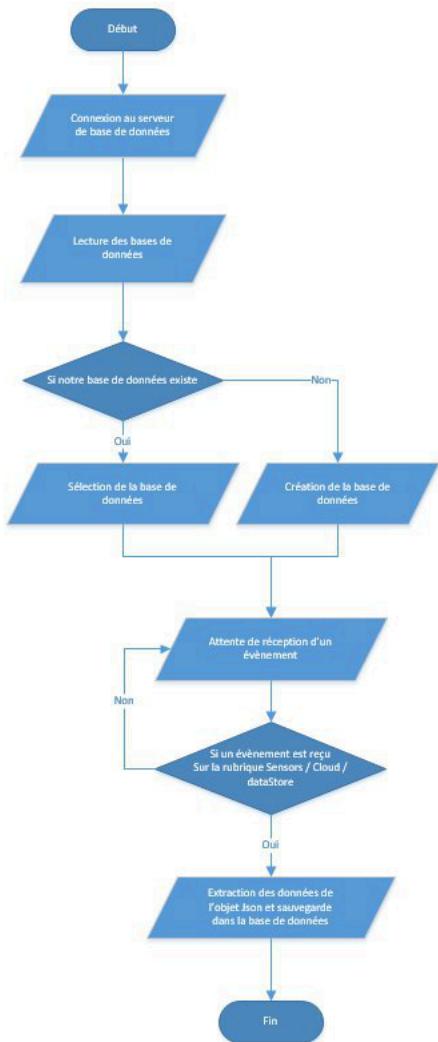


Figure 29: Diagramme de fonctionnement de notre fonction lambda (dataStore)

Tout d'abord notre fonction lambda se connecte au serveur de bases de données en local sur notre RASPBERRY PI, ensuite elle parcourt les bases de données existantes afin de sélectionner la nôtre à savoir fire_data_store. Deux cas de figure s'observent à ce niveau, si la base de données existe déjà alors notre fonction lambda la sélectionne, par contre si elle n'existe pas alors elle est créée et ensuite sélectionnée.

Cette action de création ou de sélection de notre base de données s'effectue une seule fois après l'exécution de notre fonction, car nous l'avons configuré dans la console AWS IOT à longue durée. Ainsi le contexte d'exécution reste le même, tant que l'exécution de la fonction n'est pas interrompue.

Une fois la base de données sélectionnée, notre fonction écoute sur la rubrique « sensors/cloud/dataStore » dans l'attente d'un évènement. Dès lors qu'un évènement est reçu, nous récupérons et sauvegardons les données contenues dans l'évènement au format JSON.

Pour avoir une approche simplifiée du fonctionnement de notre fonction lambda, nous avons réalisé le diagramme d'état ci-dessous montrant en trois étapes les principales opérations effectuées par celle-ci.

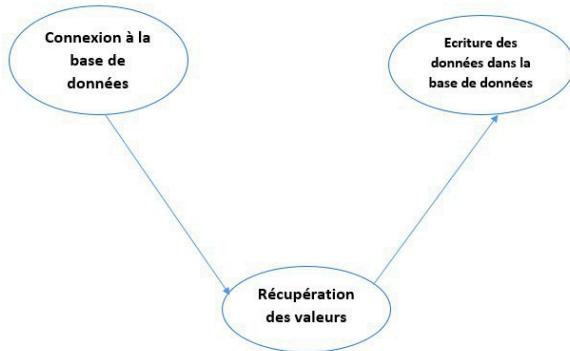


Figure 30: Diagramme d'état montrant le fonctionnement de notre fonction lambda

Après avoir effectué une pré-étude et une expérimentation de nos fonctions dans le but d'acquérir des connaissances théoriques et pratiques, nous avons adapté ces connaissances à notre problème. Étant donné que notre fonction lambda est déployée dans un noyau greengrass tournant sur notre RASPBERRY PI, il nous est impossible de communiquer avec le service AWS DYNAMODB dans le cloud afin d'y écrire nos données. C'est la raison pour laquelle nous nous sommes tournés vers une autre technologie de stockage à savoir « couchdb » qui est une base de données noSQL dont les détails sont mentionnés dans la partie I Chapitre IV section 3.

Ci-dessous se trouvent les modifications apportées à notre fonction lambda, afin de nous permettre d'écrire les valeurs reçues sur la rubrique sensors/cloud/dataStore dans notre base de données « couchdb ». Cette fonction comme nous l'avons vu dans le chapitre « concept edge computing » est déployé et exécuté sur notre noyau greengrass en local.

Afin que notre fonction lambda puisse recevoir des évènements sur la rubrique, nous l'avons abonné à la rubrique. La configuration de l'abonnement (sélection de la source et du destinataire) s'effectue dans la console AWS IOT.

Sur la capture ci-dessous, nous observons les paires clé-valeur qui seront écrites dans notre base de données.



```
66 |      # objet JSON qui sera sauvegardé dans la base de donnée couchdb
67 |      Data={
68 |          "uuid":uuid,
69 |          "time":time,
70 |          "value":value,
71 |          "type":valueType,
72 |          "location":location
73 |      }
74 |      # sauvegarde des données dans la base de données
75 |
```

Figure 31: Bout de code de notre fonction lambda montrant la structure au format JSON de nos paire clés-valeurs qui seront inscrites dans la base de données fire_data_store.

Après avoir stocké les données dans notre base de données en local, nous avons effectué une réPLICATION de notre base de données dans le cloud. Dans le but de prévenir des pertes de données.

Nous avons effectué une réPLICATION du type Master to Slave unidirectionnelle afin que si la connexion avec la base de données distante s'interrompt, nous puissions lors de la prochaine connexion répliquer uniquement les données de notre base de données locale.

IV- Réalisation de la partie traitement des données

Il est question dans cette partie de récupérer les données écrites par la fonction lambda, afin de les analyser et de détecter ainsi la présence d'un incendie ou d'un ou de plusieurs capteurs défaillants.

Ce traitement est effectué par un script python « fireSensorCheck » enregistré et configuré dans la console AWS IOT en tant qu'appareil. Nous avons choisi cette approche car nous n'avons pas trouvé dans la documentation comment connecter une fonction Lambda au service Device shadow.

Avant de pouvoir communiquer avec le service Device shadow, il est important d'authentifier notre appareil afin que la communication avec celui-ci soit chiffrée et sécurisée. Nous avons copié les fichiers d'authentification dans le même dossier que celui contenant le script, afin de pouvoir les récupérer comme arguments dans le script.

Après avoir authentifié et connecté notre appareil, nous avons implémenté la récupération et l'analyse de nos données. Ce processus est représenté dans le Diagramme ci-dessous.

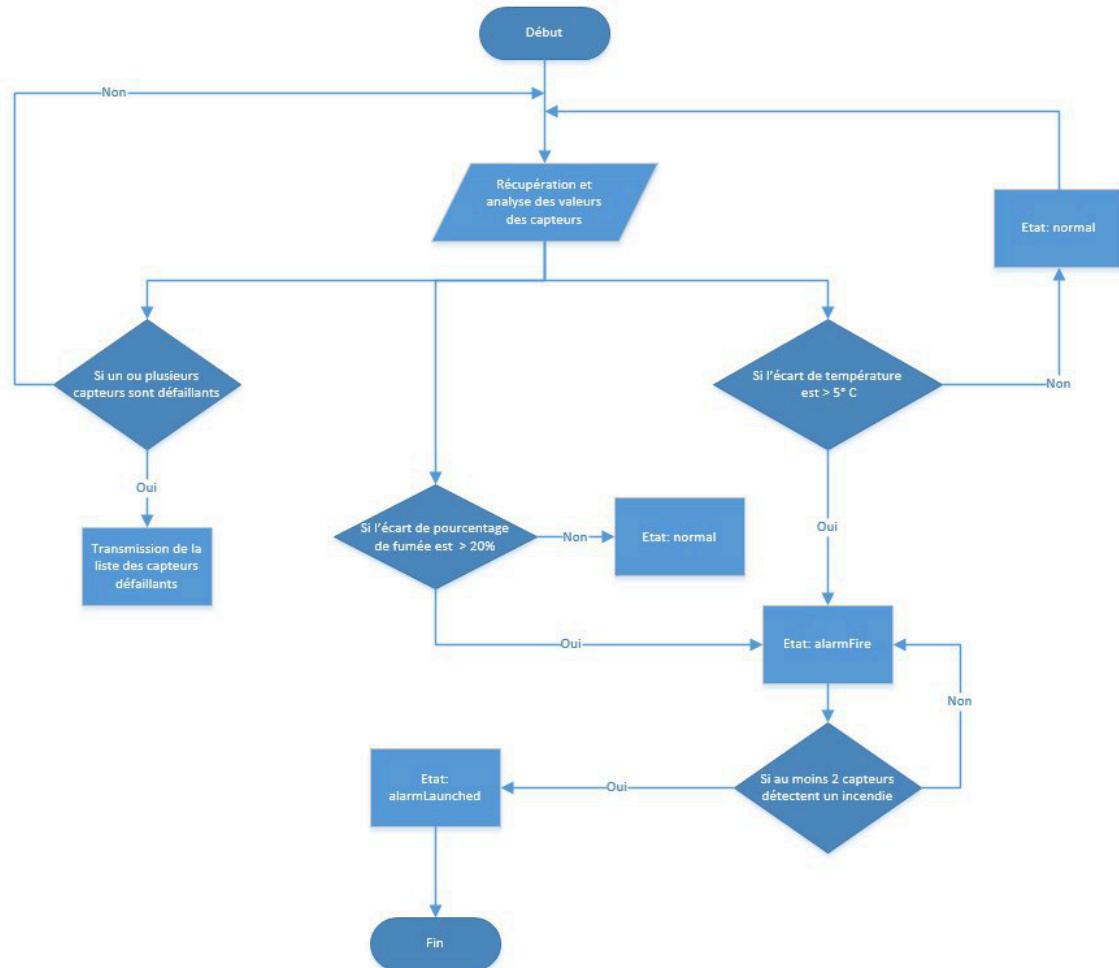


Figure 32: Diagramme montrant le fonctionnement du script `fireSensorCheck.py`

Pour effectuer notre traitement dans le but de détecter un incendie, nous récupérons uniquement les dix dernières valeurs des capteurs de température et de fumée. Ces données sont récupérées via une fonction « Mapper » implémentée dans notre base de données, qui nous permet de classer et de récupérer uniquement les données du type température et fumée depuis notre base de données. Les données récupérées en sortie du « Mapper » correspondent à une vue (design document) de notre base de données en fonctions des paramètres qui sont sélectionnés dans le « Mapper ». Cette fonction « Mapper » permet de récupérer des données spécifiques dans notre base de données et de les retourner sous forme de clé valeur.

Ensuite s'agissant des données de température, nous effectuons une différence entre la troisième et la septième valeur afin d'analyser l'écart de température entre les précédentes valeurs et les actuelles.

Si cet écart est strictement supérieur à cinq degrée Celsius, alors l'état dans le « Device shadow » est mis à jour et passe de l'état « normal » à l'état « alarmFire ». Dans le cas contraire, l'état reste inchangé (« normal »).

Si au moins deux capteurs relèvent des valeurs dont l'écart est strictement supérieur à cinq dégrée Celsius, alors l'état est mis à jour et passe de celui « alarmFire » à « alarmLaunched ». Ce qui entraîne l'envoi de l'alarme incendie au serveur de la défense civile de Dubaï.

S'agissant des données relevées par le capteur de fumée, nous effectuons une différence entre la troisième et la septième valeur afin d'analyser l'écart. Si cet écart est strictement supérieur à dix pourcents alors un incendie est détecté et l'état est mis à jour de « normal » à « alarmFire ».

La figure ci-dessous montre un exemple d'objet JSON envoyé par fireSensorCheck.py avec l'état reporté « sensorState » (correspondant à l'état du capteur) à « alarmFire » et une liste de capteur. Nous envoyons pour chaque capteur sa localisation dans la maison et son UUID.

```
# pour la détection d'une alarme
fireDict = {
    "state": {
        "reported": {
            "sensorState" : 'alarmFire',
            "sensors": [
                {
                    "uuid": "d8567b67-ecff-4ebd-865f-7ec4354fb0a",
                    "location": "001 Salon"
                }
            ]
        }
    }
}
```

Figure 33: Objet JSON envoyé dans la mise à jour de notre device shadow "fireCheck" avec un capteur localisé dans le salon correspondant à l'état reporté. Le numéro « 001 » correspond au numéro de la villa.

Pour détecter un ou plusieurs capteurs défaillants, nous comparons la liste de capteurs précédente avec celle actuelle. Si un ou plusieurs capteurs sont absents, alors ceux absents sont ajoutés à la liste de capteurs défaillants. Cette liste est ensuite mise à jour dans le device shadow « sensorCheck ». Afin de vérifier que la liste de capteurs mis à jour dans le device shadow n'envoie plus de données, nous effectuons une attente d'une minute avant d'afficher cette liste dans l'interface graphique.

Pour ne pas afficher cette liste lors de la détection par « fireSensorCheck » de capteurs défaillants, nous avons ajouté un drapeau « print » à l'objet JSON envoyé dans la mise à jour du device shadow dans le but d'afficher cette liste après l'attente. Si le drapeau est à l'état « yes » alors la liste est affichée dans l'interface graphique sinon elle ne l'est pas.

La figure ci-dessous montre un exemple d'objet JSON avec le drapeau et une liste de capteur. Nous envoyons pour chaque capteur sa localisation dans la maison.

```
# pour la détection d'un capteur défaillant
sensorDict = {
    "state": {
        "reported": {
            "print": "yes"
            "sensors": [
                "001 Salon",
                "001 Cuisine"
            ]
        }
    }
}
```

Figure 34: Objet JSON envoyé dans la mise à jour de notre device shadow "sensorCheck" avec deux capteurs localisés dans le salon et la cuisine. Le numéro « 001 » correspond au numéro de la villa.

Après avoir analyser nos données, nous avons configuré les abonnements entre notre appareil (fireSensorCheck.py) et le device shadow « fireCheck » afin que notre appareil puisse mettre à jour notre device shadow. Nous avons aussi configuré un abonnement du device shadow à notre interface graphique afin qu'elle puisse recevoir et mettre à jour l'état désiré dans le shadow. La figure ci-dessous montre un diagramme montrant le flux de données de nos devices shadow (fireCheck et sensorCheck).

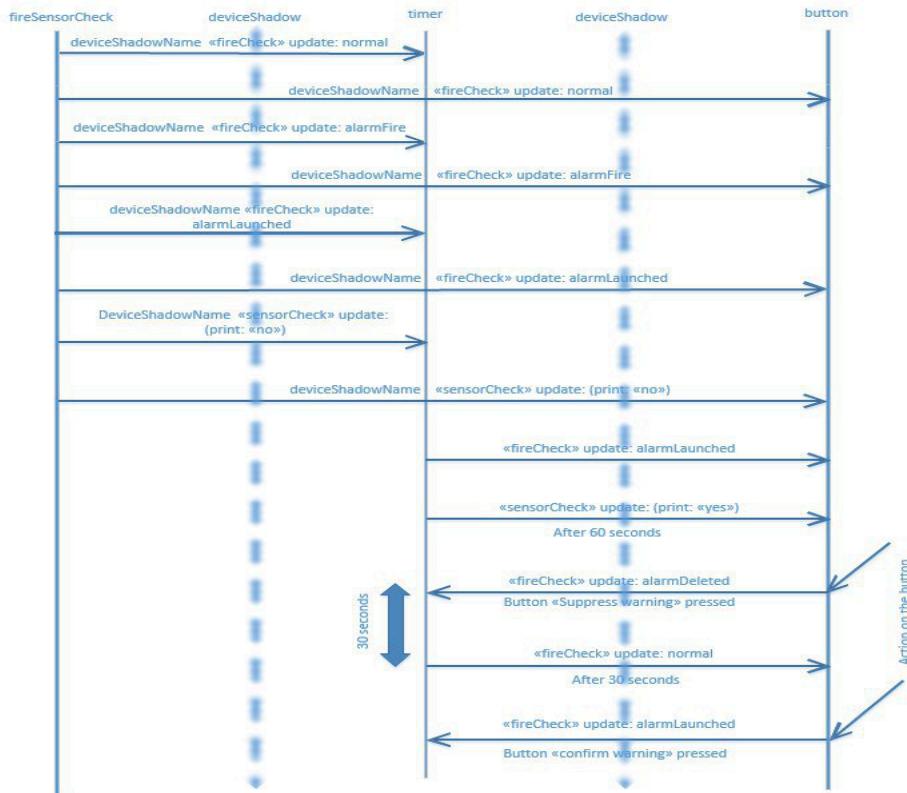


Figure 35: Flux de données du service Device Shadow (fireCheck et sensorCheck représentés par les pointillés) entre les scripts fireSensorCheck.py, timer.py et button.py.

Explication du flux de données de nos devices shadow :

Après analyse des données par le script « fireSensorCheck.py », dans le cas d'une détection d'incendie, un état reporté « normal » est publié dans le device shadow « fireCheck » (intermédiaire entre le script fireSensorCheck.py et timer.py) si aucun capteur n'a détecté un incendie. Si un incendie est détecté alors l'état est mis à jour et change de « normal » à « alarmFire ». Si au moins deux alarmes sont déclenchées alors l'état passe à « alarmLaunched ».

Dans le cas d'une détection de capteur défaillant, un état reporté est publié avec notre drapeau « print : no» dans notre device shadow « sensorcheck ». Cet état est récupéré par timer.py afin de lancer le temporisateur de soixante secondes. Une fois le temps écoulé et si la version est restée inchangée alors l'état reporté passe à « print : yes ».

Nous observons que le script fireSensorCheck.py publie trois états dans le device shadow « fireCheck » à savoir « normal », « alarmFire » et « alarmLaunched ». Ces états sont récupérés par timer.py afin de lancer un temporisateur de trente secondes s'il reçoit l'état « alarmFire » avant de passer à l'état « alarmLaunched » et si la version du device shadow n'a pas été modifiée.

Dès que les états sont récupérés dans button.py, ils sont affichés dans l’interface graphique et si l’utilisateur presse dans un premier temps sur le bouton « Suppress warning » alors l’état désiré « alarmDeleted » est publié dans le device shadow « fireCheck ». Ensuite il est récupéré par timer.py afin de lancer un temporisateur de trente secondes. Après trente secondes, cet état désiré passe à l’état « normal ».

Dans un second temps si l’utilisateur presse sur le bouton « Confirm warning » alors l’état désiré « alarmLaunched » est publié dans le device shadow « fireCheck ».

Après avoir expliqué le diagramme de flux des données de nos devices shadow, nous présentons le but et le fonctionnement du script timer.py.

Étant donné que notre script « fireSensorCheck.py » doit s’exécuter continuellement, nous avons implémenté un autre script « timer.py », afin de jouer le rôle de temporisateur après réception des états reportés et désirés à savoir « alarmFire » et « alarmDeleted » dans le cas de la détection d’incendie. Ainsi qu’après réception de l’état reporté du drapeau « print : no » et si la liste des capteurs n’est pas vide dans le cas de la détection de capteurs défaillants. L’objet JSON ci-dessous représente un exemple d’état désiré provenant de l’interface graphique.

```
# json désiré par l'utilisateur
...
fireDict = {
    "state": {
        "desired": {
            "sensorState": 'state',
            "sensors": [
                {
                    "uuid": "d8567b67-ecff-4ebd-865f-7ec4354fbb0a",
                    "location": "001 Salon"
                }
            ]
        }
    }
...}
```

Figure 36: Objet JSON envoyé dans la mise à jour de notre device shadow "fireCheck" avec un capteur localisé dans le salon correspondant à l’état désiré. Le numéro « 001 » correspond au numéro de la villa.

Le diagramme d’états ci-dessous nous montre comment fonctionne notre script timer.py.

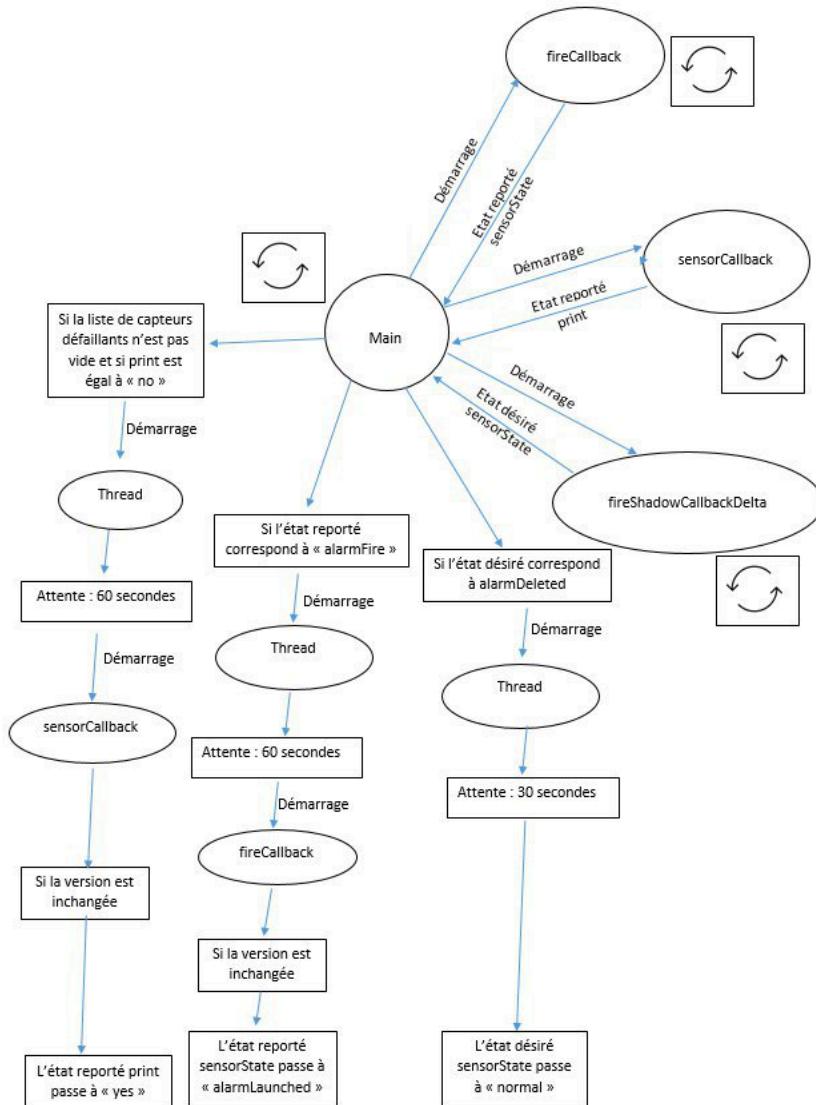


Figure 37: Diagramme d'états montrant le fonctionnement du script `timer.py`. Le cercle fléché indique que les threads s'exécutent continuellement.

Explication du fonctionnement du script `timer.py` :

Afin que notre script puisse récupérer des documents JSON et les publier, il faut au préalable qu'il puisse s'authentifier et se connecter au serveur AWS IOT. Cette authentification et connexion sont similaires à celles du script `fireSensorCheck.py`.

Après s'être authentifier, notre script plus précisément la partie principale « main » de notre code qui s'exécute en continue, démarre trois threads via des fonctions de « callback » qui vont récupérer les documents JSON contenant les états reportés et désirés dans les device shadow « `fireCheck` » et « `sensorCheck` ».

Ces threads correspondent respectivement sur la figure à « fireCallback », « sensorCallback » et « fireShadowCallbackDelta ». Ces threads écoutent continuellement en effectuant des « Get » (shadowGet correspondant à des documents JSON vide) afin de récupérer le contenu des documents JSON. Chaque thread se charge de récupérer une valeur spécifique à savoir :

- Le thread « fireCallback » récupère l'état reporté « sensorState », la liste des capteurs ayant détecté l'incendie, ainsi que le numéro de version du document JSON.
- Le thread « sensorCallback » récupère l'état reporté du drapeau « print », la liste des capteurs défaillants et le numéro de version du document JSON.
- Le thread « fireShadowCallbackDelta » récupère l'état désiré « sensorState ».

Toutes ces informations recueillies sont transmises au « main » qui va en fonction des valeurs effectuer les opérations suivantes :

- Si l'état reporté « sensorState » correspond à « alarmFire » alors un thread est lancé en lui passant comme paramètre le numéro de version actuel du document et est endormi pendant trente secondes. Une fois le décompte effectué, ce thread va lancer un autre thread « fireCallback », afin de récupérer la version du document JSON après trente secondes. Si la version du document est restée inchangée alors l'état reporté est changé de « alarmFire » à « alarmLaunched ». Par contre si la version du document est changée alors le thread est supprimé.
- Si la liste des capteurs défaillants n'est pas vide et que le drapeau est à la valeur « print : no » alors un thread est lancé avec le numéro de version en paramètre et est endormi pendant soixante secondes. Après le décompte, ce thread va lancer un autre thread « sensorCallback » qui va récupérer la version du document JSON et le comparer avec celui passé en paramètre. Si la version est la même, alors la valeur du drapeau dans le device shadow passe à « print : yes » sinon le thread est supprimé.
- Si l'état désiré « sensorState » correspond à « alarmDeleted » alors un autre thread est lancé cette fois sans paramètre et est endormi pendant trente secondes. Une fois le décompte écoulé, ce thread va juste mettre à jour l'état désiré de « alarmDeleted » à « normal ».

Pour mieux comprendre le processus de lancement des threads par notre thread « main », nous avons réalisé le diagramme de séquence ci-dessous.

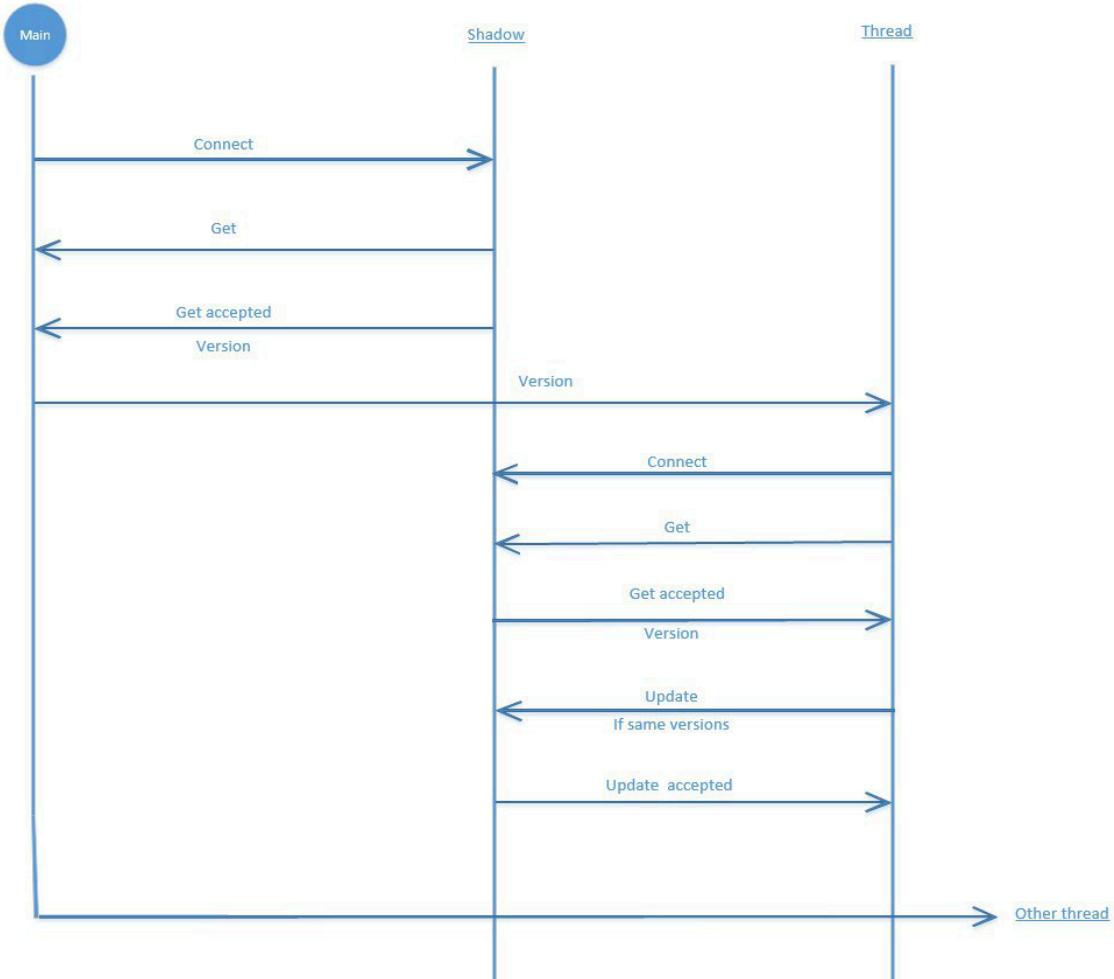


Figure 38: Diagramme de séquences montrant ainsi le processus de lancement de chaque thread

Le diagramme ci-dessus fonctionne comme suit :

- Le thread « main » se connecte à notre mini cloud, ce qui lui permet de récupérer et de mettre à jour les documents JSON contenus dans les devices shadows. Une fois connecté, il effectue un « Get » auprès des device shadows afin de récupérer le document JSON contenu dans chaque device. Une fois la requête effectuée, le device shadow lui répond avec un statut de réponse « Get accepted » montrant ainsi que la requête a abouti et lui envoyant la version du document JSON courant.
- Dès que le document JSON est reçu avec le numéro de version, le thread « main » va lancer un thread en lui passant le numéro de version actuel reçu du device shadow, qui va à son tour effectuer des opérations, ensuite se connecter et effectuer de nouveau un « Get » sur le device shadow. Une fois le document JSON reçu avec le numéro de version obtenu, ce thread va comparer le numéro de version reçu après son « Get » avec celui passé en paramètre par le thread « main ».

- Si les versions sont identiques alors ce thread (celui lancé par le « main ») va effectuer une mise à jour du document JSON contenu dans le device shadow, sinon le thread est supprimé.
- La flèche quittant du thread « main » à « Other thread » permet juste d'indiquer que d'autres threads sont lancés soit par le thread « main » soit par le thread lancé par celui-ci.

V- Réalisation de la partie affichage

Il est question dans cette partie de créer une interface graphique, afin de permettre à l'utilisateur de consulter et de mettre à jour l'état du document JSON contenu dans le device shadow. L'utilisateur est notifié lors de chaque changement d'état apporté dans le device shadow, afin de gérer les alarmes incendies.

Pour réaliser cette interface graphique utilisateur, nous avons utilisé le motif MVC (Modèle-Vue-Contrôleur). Ce motif a pour but de dissocier la partie affichage de nos données correspondant à la « Vue » (V), de la partie interaction avec l'utilisateur correspondant au « Contrôleur » (C) et de la partie récupération des données correspondant à notre « Modèle » (M). Pour avoir plus de détail sur le motif MVC, veuillez consulter [2.5.1].

Afin de mieux comprendre notre « design pattern » MVC nous avons réalisé le diagramme ci-dessous.

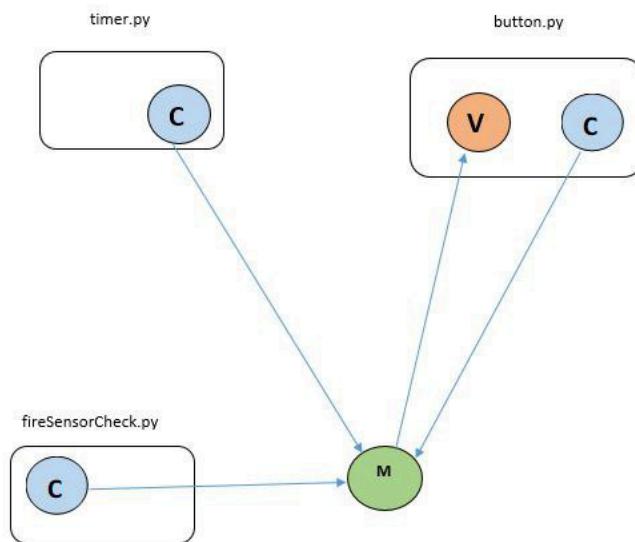


Figure 39: « Design Pattern » MVC utilisé pour la réalisation de notre interface graphique avec « M » pour Modèle, « C » pour Contrôleur et « V » pour Vue.

Le « Modèle » dans notre « design pattern MVC » correspond au document JSON de notre device shadow qui est mis à jour par nos scripts « fireSensorCheck.py », « timer.py » et « button.py » qui correspondent aux « Contrôleurs ». Les modifications apportées par les

« Contrôleurs » sur le « Modèle » sont affichées dans la « Vue » qui correspond à notre interface graphique.

Une fois que notre motif MVC a été réalisé, nous avons implémenté la « Vue » (interface graphique) qui affiche les données du document JSON contenu dans les device shadow et permet la gestion des alarmes par l'utilisateur. Cette modification effectuée par l'utilisateur correspond à l'état désiré dans le document JSON du device shadow.

Nous avons effectué la simulation pour une villa mais notre interface peut recevoir huit alertes maximums. La figure ci-dessous présente notre interface graphique. Nous allons effectuer nos différents scénarios dans la partie encadrée pour une villa.

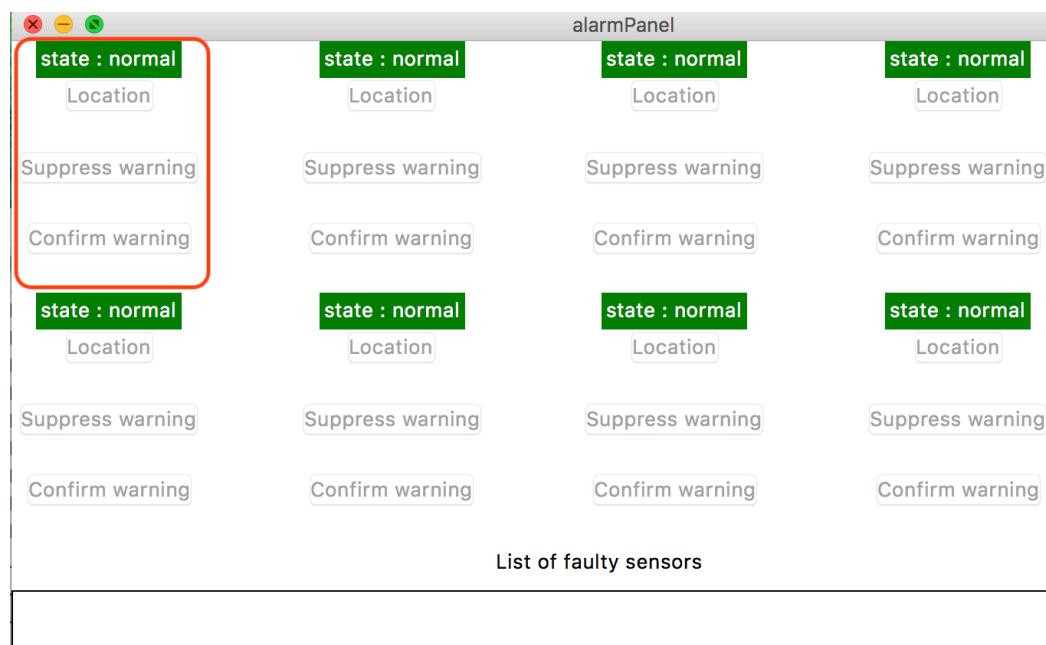
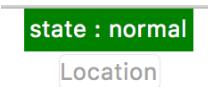


Figure 40: Interface graphique pour interagir avec l'utilisateur, afin qu'il puisse gérer les alarmes.

Les captures ci-dessous montrent plusieurs scénarios d'interactions avec un utilisateur en local :

- Premier scénario pendant lequel aucune alarme n'est détectée, dans ce cas l'état correspond à « normal » et l'utilisateur ne peut pas dans ce cas interagir avec l'interface graphique car les boutons sont désactivés.



SUPPRESS WARNING

CONFIRM WARNING

Figure 41: État "normal" avec aucune possibilité d'interaction avec l'utilisateur dans le cas de la gestion d'alarmes

- Deuxième scénario pendant lequel une alarme est détectée, l'état correspond à « alarmFire » et ce n'est uniquement dans ce cas de figure que l'utilisateur peut interagir avec l'interface graphique.



SUPPRESS WARNING

CONFIRM WARNING

Figure 42: Etat "alarmFire" dans lequel l'utilisateur peut interagir avec l'interface graphique

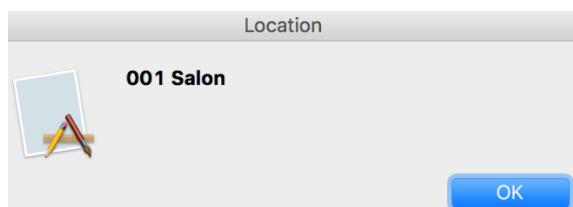


Figure 43: Bouton "Location" pressé par l'utilisateur affichant ainsi la position du capteur ayant détecté l'incendie dans la villa.

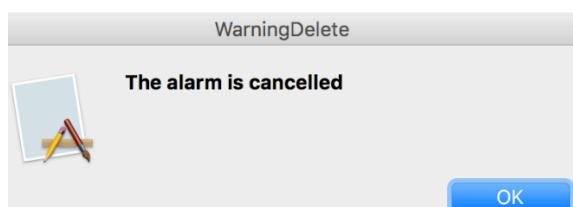


Figure 44: Bouton "Suppress warning" pressé par l'utilisateur ce qui a pour conséquence l'annulation de l'alarme.

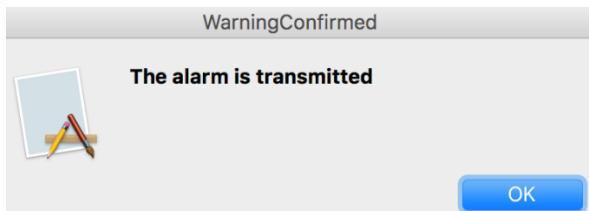


Figure 45: Bouton "Confirm warning" pressé par l'utilisateur ayant pour conséquence la confirmation de l'alarme.

- Troisième scénario dans lequel plusieurs alarmes sont détectées par au moins deux capteurs ou alors l'utilisateur a pressé sur le bouton « Confirm warning ». Dans ce cas l'utilisateur peut uniquement consulter la localisation des capteurs ayant détectés l'incendie.

state : alarmLaunched

Location

Suppress warning

Confirm warning

Figure 46: État "alarmLaunched" dans lequel l'utilisateur peut uniquement observer la liste des capteurs ayant détectés l'incendie.

- Quatrième scénario dans lequel un ou plusieurs capteurs sont défaillants, dans ce cas la liste de ces capteurs est affichée dans la fenêtre déroulante ci-dessous :

List of faulty sensors

001 Salon

Figure 47: Affichage des capteurs défaillants dans la liste déroulante. Dans notre situation nous avons un capteur défaillant localisé dans le salon de la villa numéro 1.

La capture ci-dessous montre notre interface graphique dans le cloud sur une instance EC2. Pour qu'un utilisateur puisse se connecter à distance et utiliser notre interface graphique pour gérer les alarmes, il faut qu'il établisse une connexion en se connectant à un serveur VNC (Virtual Network Computing). Après s'être connecté l'utilisateur peut interagir avec l'interface graphique comme s'il était en local. Nous avons réutilisé la même interface que celle utilisée en local.

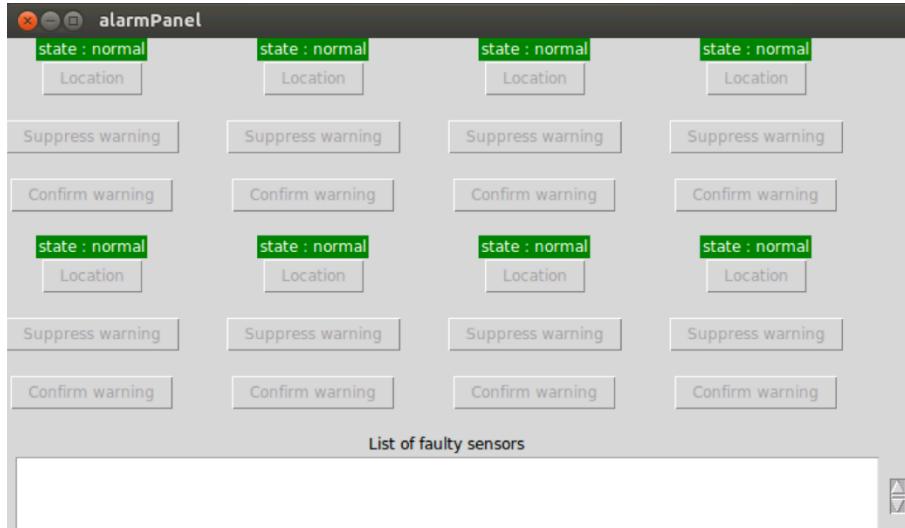


Figure 48: Interface graphique dans le cloud sur une instance EC2 avec un système Ubuntu 16.04.

VI- Bugs

Nous avons rencontré un problème quant à la gestion des threads, plus précisément dans la réalisation de la partie affichage. Ce problème entraîne un crash de l'application. Étant donné que plusieurs threads effectuent des requêtes simultanément, nous stockons ces différentes requêtes dans une queue afin de gérer la concurrence. Mais cela a pour conséquence un ralentissement de l'exécution de notre programme ainsi qu'un dysfonctionnement lors de l'affichage.

Pour palier à ce problème, une éventualité serait de lancer un nombre limité de threads accédant à la ressource et protéger l'accès à la rubrique concurrentielle (canal partagée par tous les threads). Ceci a pour conséquence de rendre notre exécution asynchrone qui ne répond pas à nos attentes.

VII- Développements et améliorations futures

Étant donné que notre solution est un prototype, quelques développements et améliorations peuvent être apportés afin de rendre notre solution utilisable en entreprise. Nous pouvons citer comme améliorations :

- La création d'une application web pour qu'un utilisateur puisse se connecter, visualiser les données en temps réel et gérer les alarmes. Cela peut être réalisé en utilisant des bibliothèques existantes comme ReactJS et AngularJS.
- Étant donné que les adresses IP attribuées à nos RASPBERRY PI sont dynamiques et non statiques, il faudrait utiliser un switch Ethernet afin de parer à ce problème. Ce problème entraîne le changement d'adresse IP dans la fonction lambda afin qu'elle puisse accéder à la base de données en local. Par contre nous avons utilisé le programme « bonjour » afin de nous connecter à nos RASPBERRY PI sans avoir à connaître leurs adresses IP mais cette solution n'est pas fonctionnelle en utilisant des fonctions Lambda même déployés sur notre mini cloud local.
- Sachant que le traitement de nos données ne s'effectue pas par une fonction Lambda mais par un script en local, il serait intéressant d'implémenter une solution en utilisant ces fonctions afin que notre traitement puisse être déployé dans le cloud en utilisant la même fonction lambda.
- Gérer l'accès concurrentiel à la rubrique du device shadow, dans le but de rendre notre programme évolutif.

CONCLUSION

Arrivé au terme de notre travail, nous sommes satisfaits du travail effectué et nous avons pris beaucoup de plaisir à nous familiariser avec les services cloud fournis par Amazon à savoir AWS LAMBDA, AWS EC2, AWS GREENGRASS, AWS IOT et AWS DYNAMODB. Nous avons particulièrement aussi apprécié la puissance, la performance et le faible cout financier lors de l'utilisation des fonctions Lambda. Sans oublier le service Device Shadows qui permet de créer un objet virtuel de notre appareil afin de pouvoir communiquer l'état d'un appareil avec d'autres appareils, des applications et le cloud de manière totalement sécurisée. Nous avons compris comment est gérée la sécurité des communications entre les objets connectés et d'autres services AWS ou le cloud. Nous nous sommes familiarisés avec la nouvelle technologie de base de données NoSQL plus précisément la base de données CouchDB avec laquelle nous n'avons pas eu du mal à comprendre son utilisation. Étant donné que toute notre solution est implémentée en utilisant le langage Python, nous avons avec satisfaction maîtrisé son fonctionnement et son utilisation. Cependant notre solution multi-threadé n'est pas satisfaisante car elle entraîne un dysfonctionnement de la partie affichage. Malheureusement nous étions à court de temps et nous n'avons pas pu débugger ce problème.

Il était question durant la réalisation de notre travail de déployer le traitement et le stockage des données issues d'objets interconnectés qui s'effectuaient dans le cloud en local. Ayant trouvé une solution à la problématique posée, nous sommes fiers d'avoir réalisé un prototype fonctionnel à quatre-vingt pourcents et d'avoir montré que ce traitement et stockage des données peuvent être déployés en local. Tout cela a été rendu possible par l'utilisation des services proposés par AWS IOT qui nous a permis de créer un mini cloud en local afin d'y exécuter nos fonctions et connecter nos équipements.

Cependant plusieurs points restent encore à améliorer quant à l'affichage des données de capteurs en temps réels via une application web afin que la solution soit utilisable en entreprise, d'utiliser switch Ethernet afin d'établir une connexion locale avec nos appareils, d'effectuer le traitement de nos données par une fonction Lambda et enfin de résoudre le problème de l'accès concurrentiel par les threads afin d'avoir un prototype fonctionnel en local et dans le cloud.

BIBLIOGRAPHIE

[Partie. Chapitre. Numéro]. Nom de l'auteur, titre de la page, URL, Année, date d'accès

[1.3.1]. Amazon Web Services, Création d'une fonction lambda simple, https://docs.aws.amazon.com/fr_fr/lambda/latest/dg/get-started-create-function.html, 2018, accédé en Mars 2018.

[1.3.2]. Amazon Web Services, Accès à DynamoDB, https://docs.aws.amazon.com/fr_fr/amazondynamodb/latest/developerguide/AccessingDynamoDB.html, 2018, accédé en Mars 2018.

[1.3.3]. Anish Borkar, AWS-Greengrass-Samples, <https://github.com/aws-samples/aws-greengrass-samples/blob/master/traffic-light-example-python/carAggregator.py>, 15 Novembre 2017, accédé en Mars 2018.

[1.3.4]. Amazon Web Services, Configuration des rôles IAM, https://docs.aws.amazon.com/fr_fr/greengrass/latest/developerguide/config-iam-roles.html, 2018, accédé en Avril 2018.

[1.3.5]. Nilesh Suryavanshi, What are cloud computing services, <https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e>, 8 Novembre 2017, accédé en Mars 2018.

[1.3.6]. Nilesh Suryavanshi, What are cloud computing services, <https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e>, 8 Novembre 2017, accédé en Mars 2018.

[1.3.7]. Amazon Web Services, Sources d'événements prises en charge, https://docs.aws.amazon.com/fr_fr/lambda/latest/dg/invoking-lambda-function.html, 2018, accédé en Mars 2018.

[1.3.8]. François Quellec, Pierre-Samuel Rochat, Bryan Perroud, Emmanuel Schmid, AWS Lambda, <https://cyberlearn.hes-so.ch/mod/folder/view.php?id=820074>, 2018, accédé en Juillet 2018.

[1.4.1]. Wikipedia, Raspberry Pi, https://fr.wikipedia.org/wiki/Raspberry_Pi, 11 Avril 2018, accédé en Avril 2018.

[1.4.2]. Amazon Web Services, Création de fonctions lambda, https://docs.aws.amazon.com/fr_fr/lambda/latest/dg/lambda-app.html, 2018, accédé en Avril 2018.

[1.4.3]. Blogger, Top programming languages used in IoT, <http://www.iot.qa/2018/01/top-programming-languages-used-in-iot.html>, 17 Janvier 2018, accédé en Avril 2018.

[1.4.4]. J. Chris Anderson, Jan Lehnardt, Noah Slater, « Pourquoi CouchDB ? » du livre « CouchDB: The Definitive Guide », <http://guide.couchdb.org/editions/1/fr/why.html>, 26 Janvier 2010, accédé en Mai 2018.

[1.4.5]. Solid IT, DB-Engines, System properties comparison CouchDB vs. MongoDB, <https://db-engines.com/en/system/CouchDB;MongoDB>, 2018, accédé en Mai 2018.

[1.4.6]. DB-Engines, Maitrisez les bases de données NoSQL <https://openclassrooms.com/courses/maitrisez-les-bases-de-donnees-nosql/et-les-graphes-dans-tout-ca>, 2018, accédé en Mai 2018.

[1.4.7]. raspberrypi, raspberry pi 3 modèle B, <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>, 2018, accédé en Juillet 2018.

[1.4.8]. dexterindustries, Kit de démarrage grove pi+ pour raspberry pi, <https://www.dexterindustries.com/shop/grovepi-starter-kit-2/>, 2018, accédé en Juillet 2018.

[1.4.9]. Ecran tactile LCD 5 pouces, [https://fr.aliexpress.com/item/raspberry-pi-2-Touch-Screen-LCD-5-inch-Resistive-Touch-Screen-LCD-HDMI-interface/32314327368.html/](https://fr.aliexpress.com/item/raspberry-pi-2-Touch-Screen-LCD-5-inch-Resistive-Touch-Screen-LCD-HDMI-interface/32314327368.html), accédé en Juillet 2018.

[1.5.1]. Amazon Web Services, Démarrez avec AWS Greengrass, https://docs.aws.amazon.com/fr_fr/greengrass/latest/developerguide/gg-gs.html, 2018, accédé en Mai 2018.

[1.5.2]. Dave Lage, Dirkjan Ochtman, couchdb-python, <https://github.com/djc/couchdb-python>, 16 Fevrier 2018, accédé en Mai 2018.

[1.5.3]. Amazon, rubriques shadow MQTT, https://docs.aws.amazon.com/fr_fr/iot/latest/developerguide/device-shadow-mqtt.html, 2018, accédé en Juillet 2018.

[2.1.1]. Wikipédia, MQTT, <https://fr.wikipedia.org/wiki/MQTT>, 2018, accédé en Juillet 2018.

[2.5.1]. Nicolas Hilaire, Le Pattern MVC, <https://openclassrooms.com/fr/courses/1730206-apprenez-asp-net-mvc/1730466-le-pattern-mvc>, 23 Novembre 2017, accédé en Juillet 2018.

AUTHENTIFICATION

Je confirme avoir réalisé mon travail individuellement et de n'avoir pas utilisé d'autres sources que celles mentionnées dans la bibliographie.